Memory Management with `mmap`

What if we use **`mmap`** instead of **`malloc`** always?

✗ Wasteful — ***low utilization***

need 16 bytes, get 4096

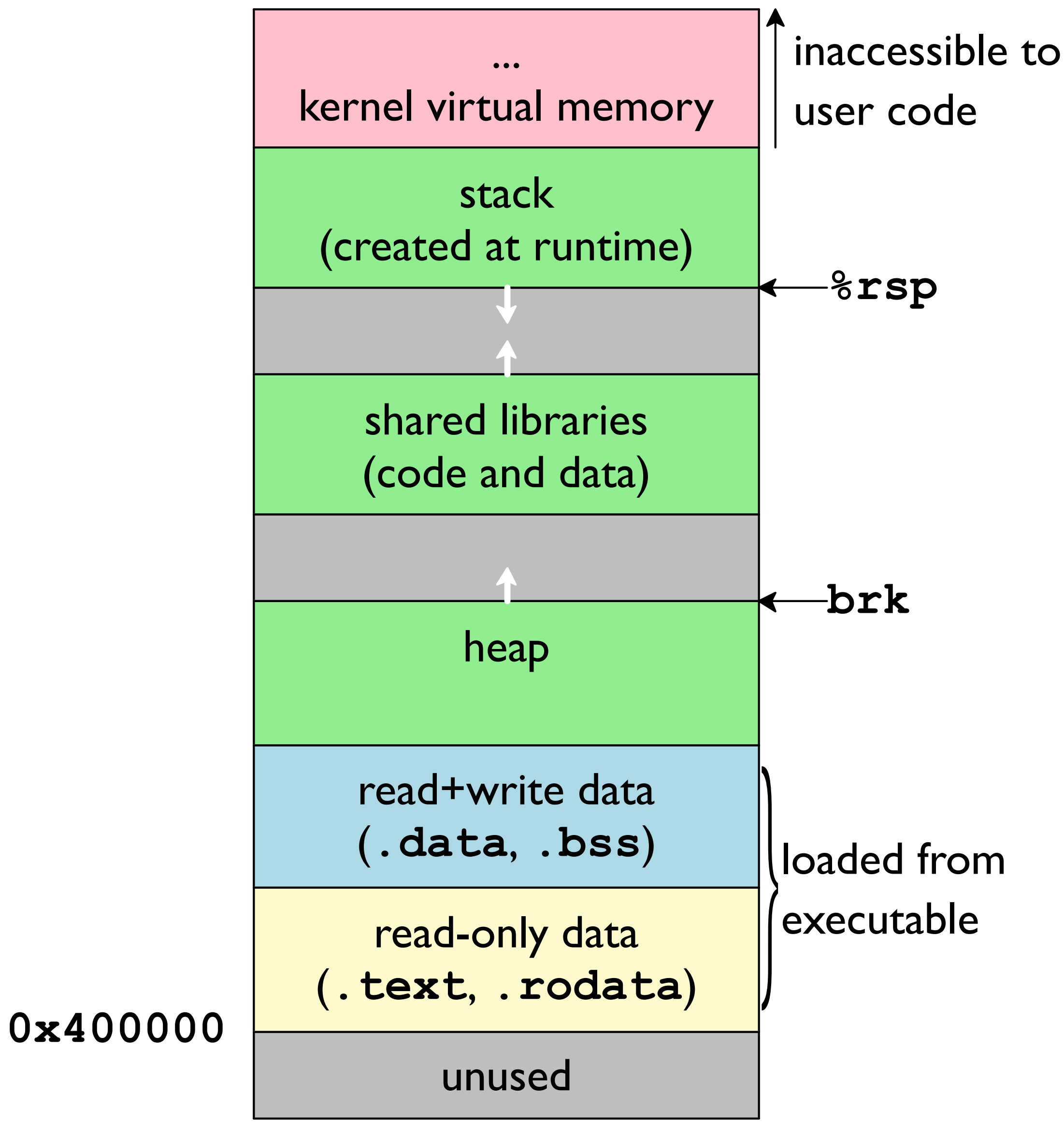
✗ Slow — ***low throughput***

have to interact with kernel every time,
and those 4096 bytes are all zeroed

✗ Complicated

have to remember the size to unmap

Process Memory Layout



Memory Management with `sbrk`

```
#include <unistd.h>

void *sbrk(intptr_t increment);
```

Grows the ***program break***, a.k.a. **`brk`**, and returns the old program break

Effectively, allocates **`increment`** bytes

Do not use **`sbrk`** in a program that also uses **`malloc`** or anything that calls **`malloc`** (such as **`printf`**)

Memory Management with `sbrk`

What if we use **`sbrk`** instead of **`malloc`** always?

✓ Economical — ***good utilization***, at first

need 16 bytes, get 16

✗ Somewhat slow — ***somehat low throughput***

have to interact with kernel every time

✗ Complicated

have to remember the size to unsbrk(?)

✗ Inexpressive — ***low utilization*** when done with data

at best, can free last chunk allocated

Standard C Allocation

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *p);

void *calloc(size_t count, size_t size);
void *realloc(void *p, size_t new_size);
```

`malloc` allocates at least **`size`** bytes

`free` accepts a pointer (just once) from **`malloc`**

behind the scenes: **`mmap`** or **`sbrk`**, maybe **`munmap`**

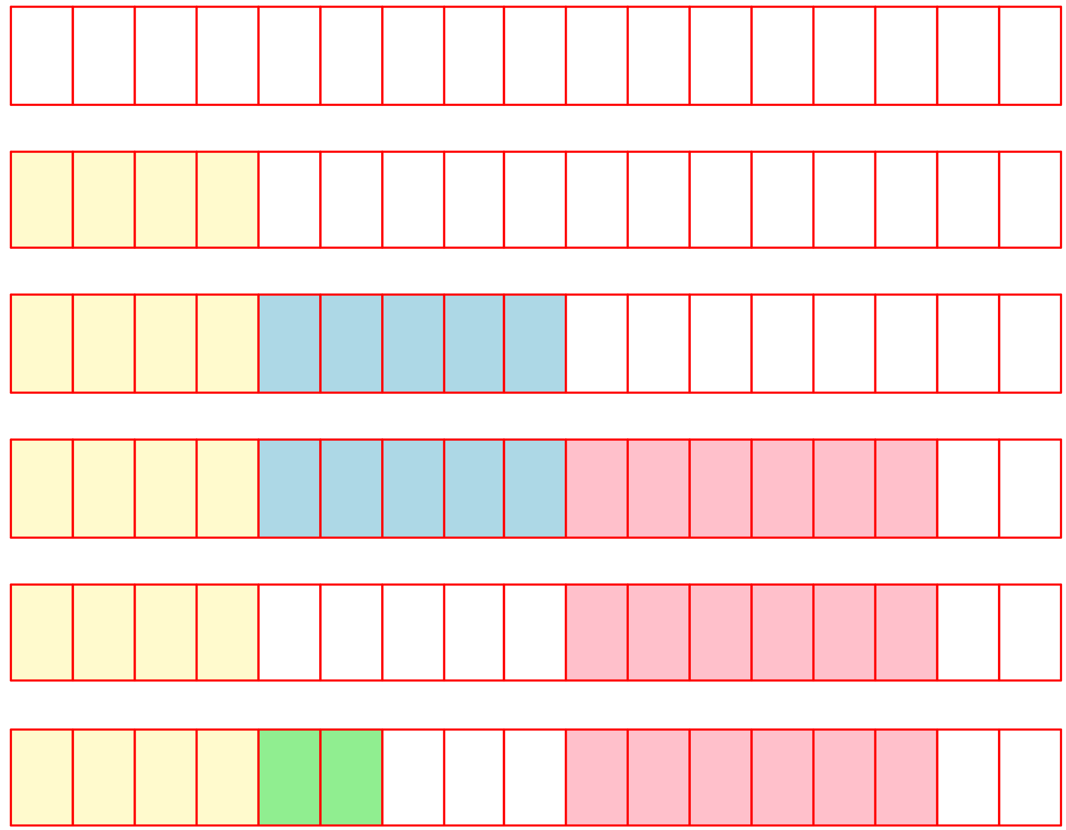
`calloc` is multiply, then **`malloc`**, then **`bzero`**

`realloc` is **`malloc`**, then **`memcpy`**, then **`free`**

maybe with a shortcut

Allocation Example

```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)
```



Allocation: Application Side

Rights:

- Call freely interleave **`malloc`** and **`free`**

Responsibilities:

- Must write to only allocated (not-yet-freed) blocks
- Must call **`free`** only once on each **`malloc`** result
- Must call **`free`** enough to limit memory use

Allocation: Allocator Side

Rights:

- Can pick arbitrary virtual addresses

within alignment constraints

Responsibilities:

- Must accept any size request
- Must accept any number of requests
- Must return non-overlapping blocks
- Must not write to allocated (not-yet-freed) blocks
- Must respond immediately (i.e., can't reorder requests)

Allocation: Performance Goals

Utilization — use as few pages as possible

$$\text{measure as } \frac{\text{aggregate payload}}{\text{pages used}}$$

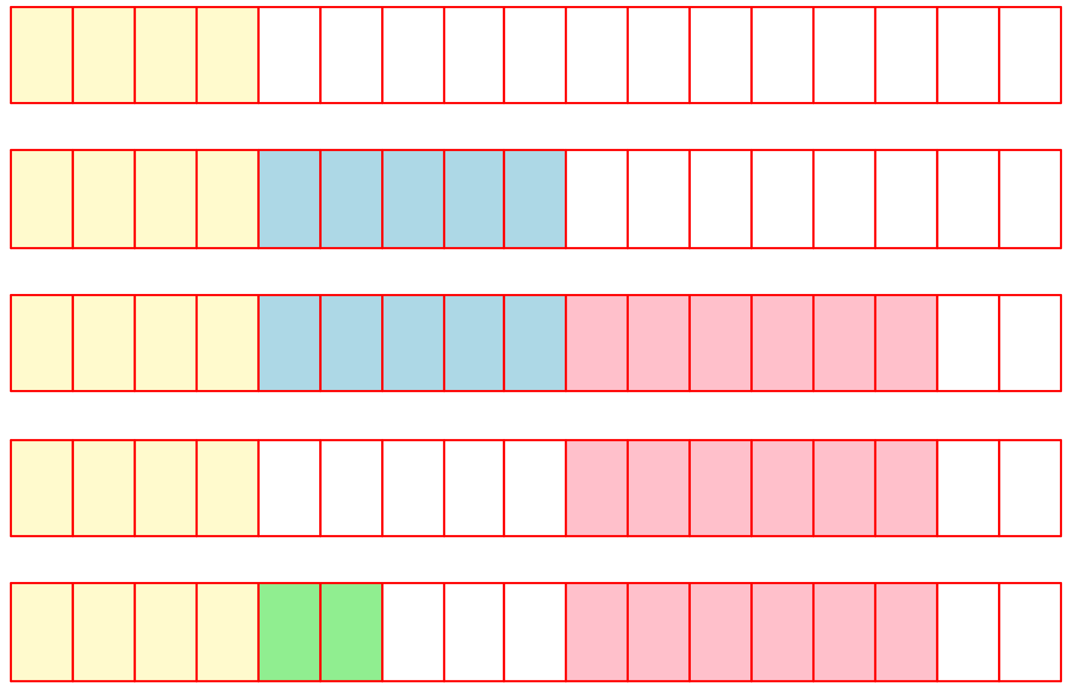
- `malloc(`n`)` ⇒ ***payload*** size n
- Sum of n not yet `freed` = ***aggregate payload***

Throughput — `malloc`/`free` as fast as possible

$$\text{measure as } \frac{\text{number of operations performed}}{\text{seconds used}}$$

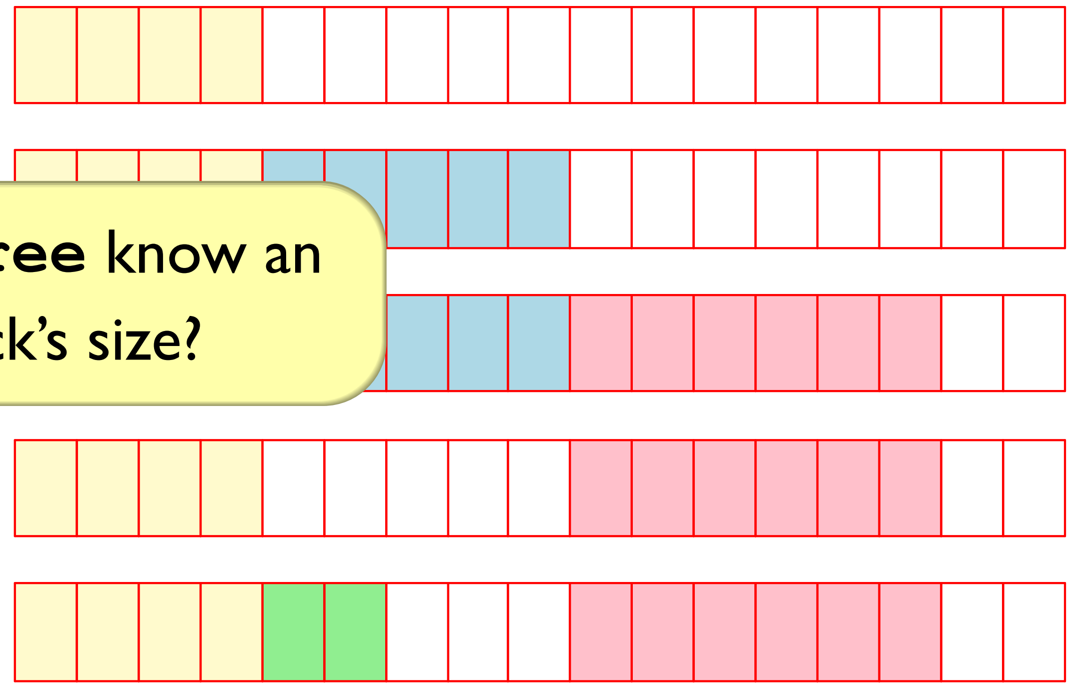
Allocator Design Questions

```
p1 = malloc(4)
p2 = malloc(5)
p3 = malloc(6)
free(p2)
p4 = malloc(2)
```



Allocator Design Questions

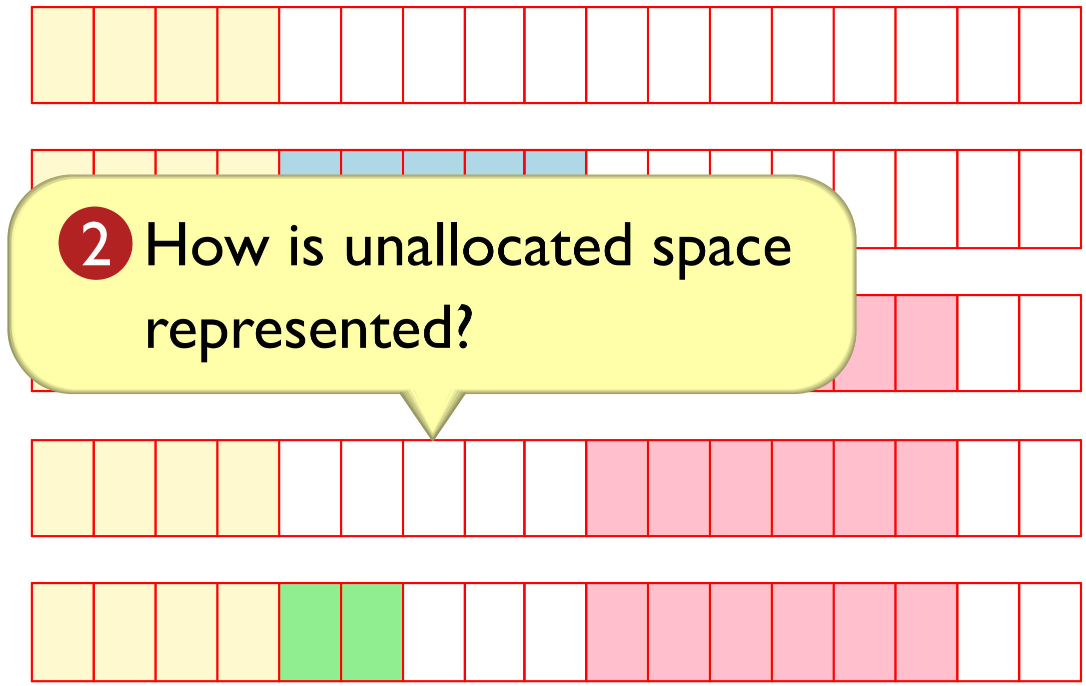
```
p1 = malloc(4)

p2 =

p3 =

free(p2)

p4 = malloc(2)
```



❗ How does **`free`** know an allocated block's size?

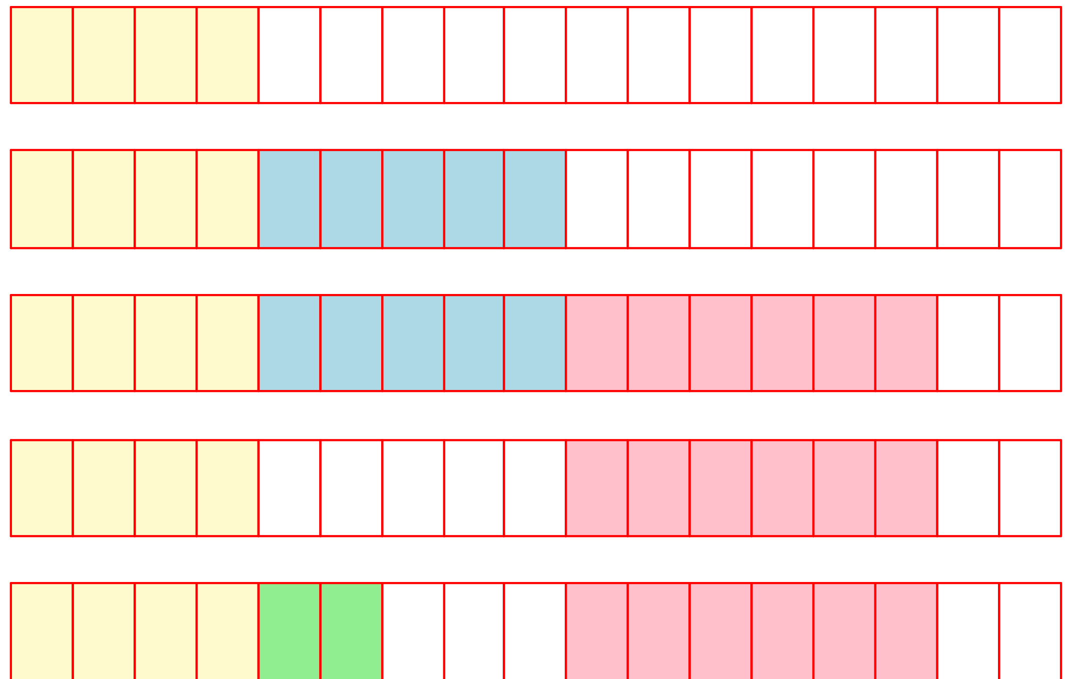
Allocator Design Questions

```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)
```



Allocator Design Questions

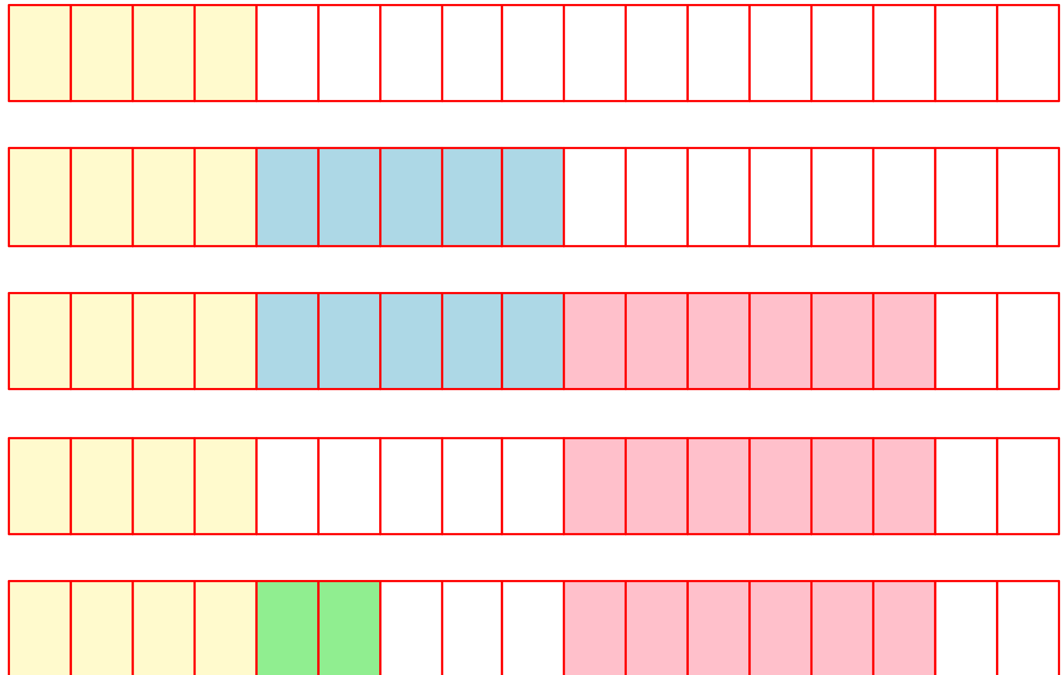
```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)
```



3 How is unallocated space selected for each allocation?

Allocator Design Questions

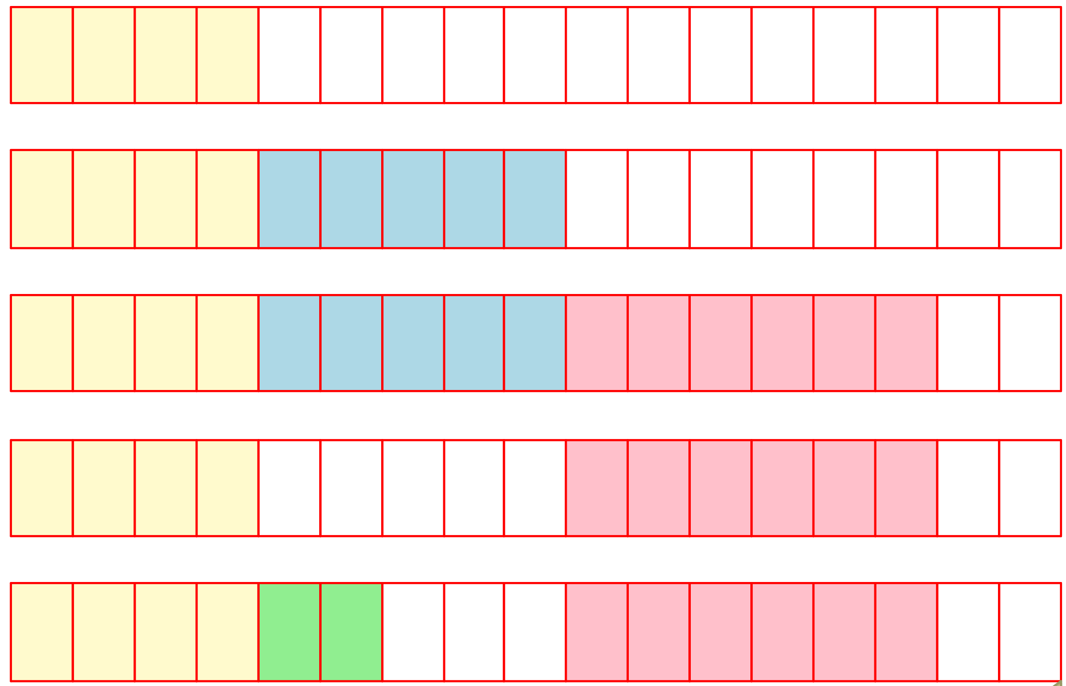
```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)
```



4 How finely is unallocated space tracked?

Allocator Design Questions

```
p1 = malloc(4)
p2 = malloc(5)
p3 = malloc(6)
free(p2)
p4 = malloc(2)
```



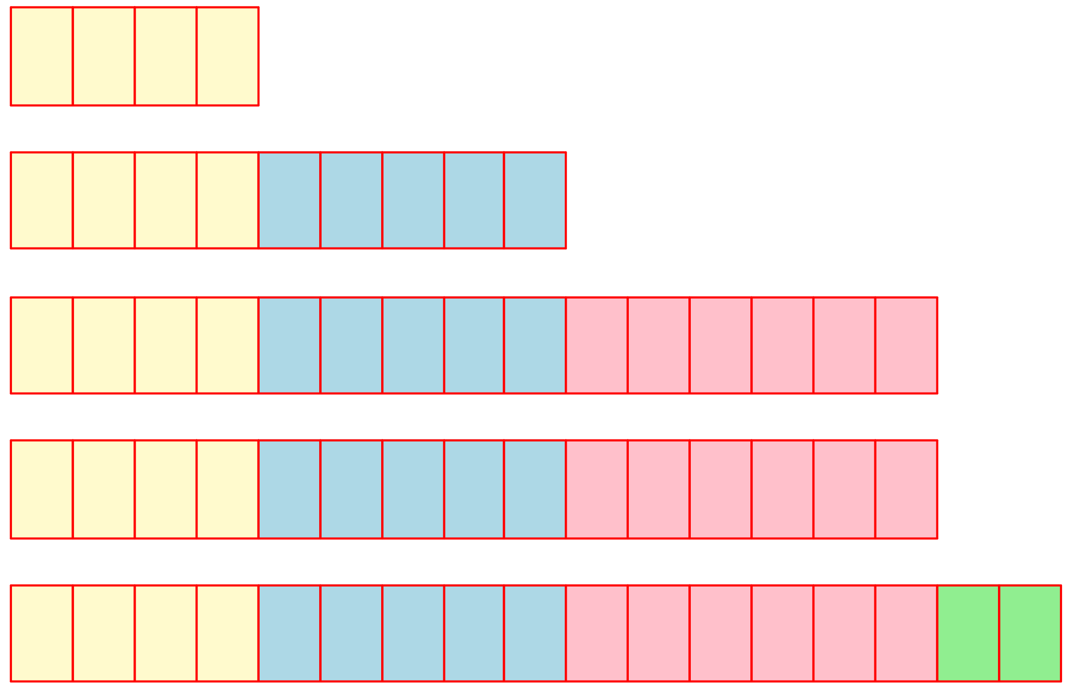
5 When are more pages needed from the kernel?

Allocator Design Questions

1. How does `free` know an allocated block's size?
2. How is unallocated space represented?
3. How is unallocated space selected for each allocation?
4. How finely is unallocated space tracked?
5. When are more pages needed from the kernel?

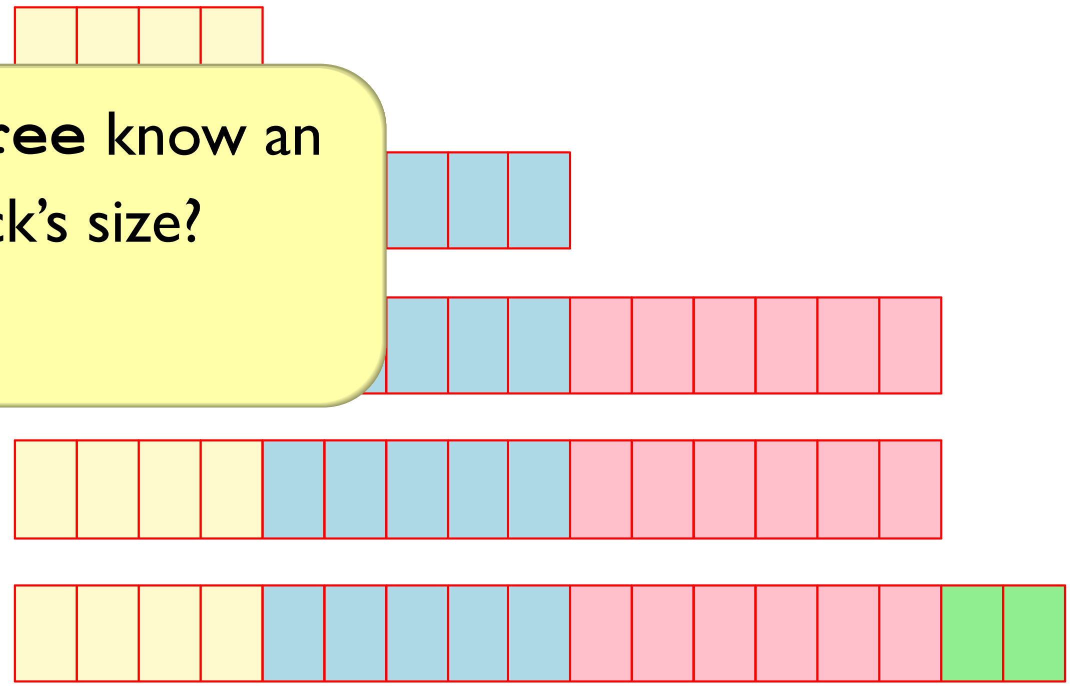
Naive `sbrk` Allocator

```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)
```



Naive `sbrk` Allocator

```
p1 = malloc(4)

p2 =

p3 =

free(p2)

p4 = malloc(2)
```

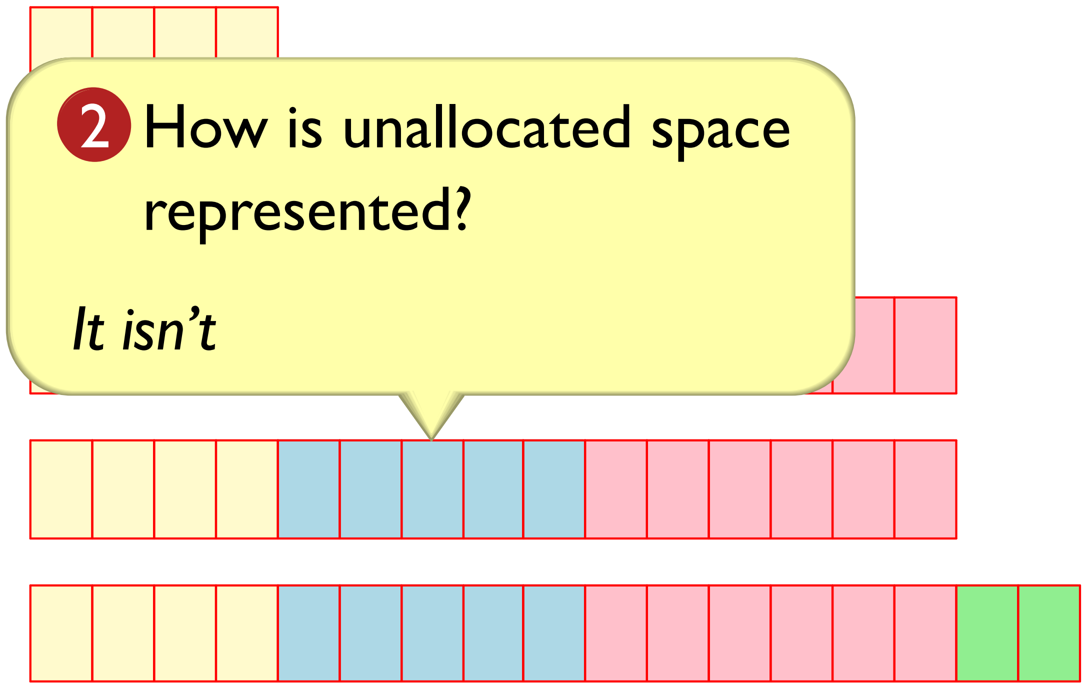


How does **`free`** know an allocated block's size?

It doesn't

Naive `sbrk` Allocator

```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)
```

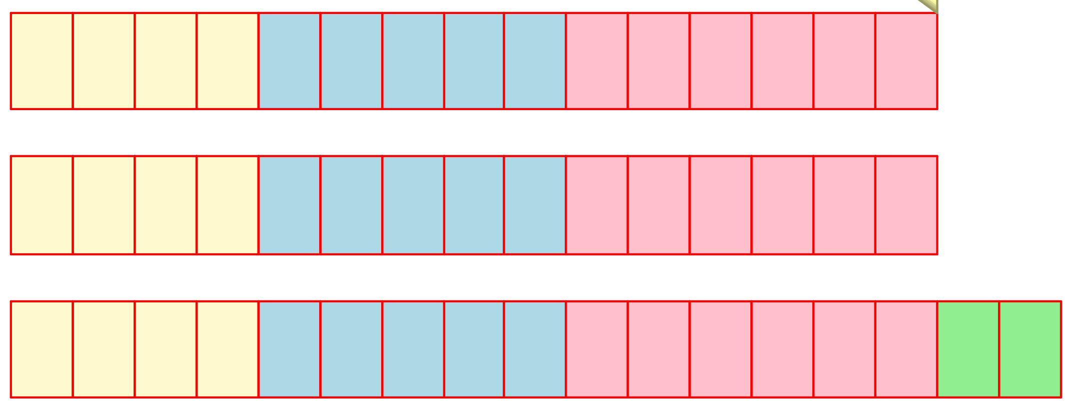


Naive `sbrk` Allocator

```
p1 = malloc(4

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)
```

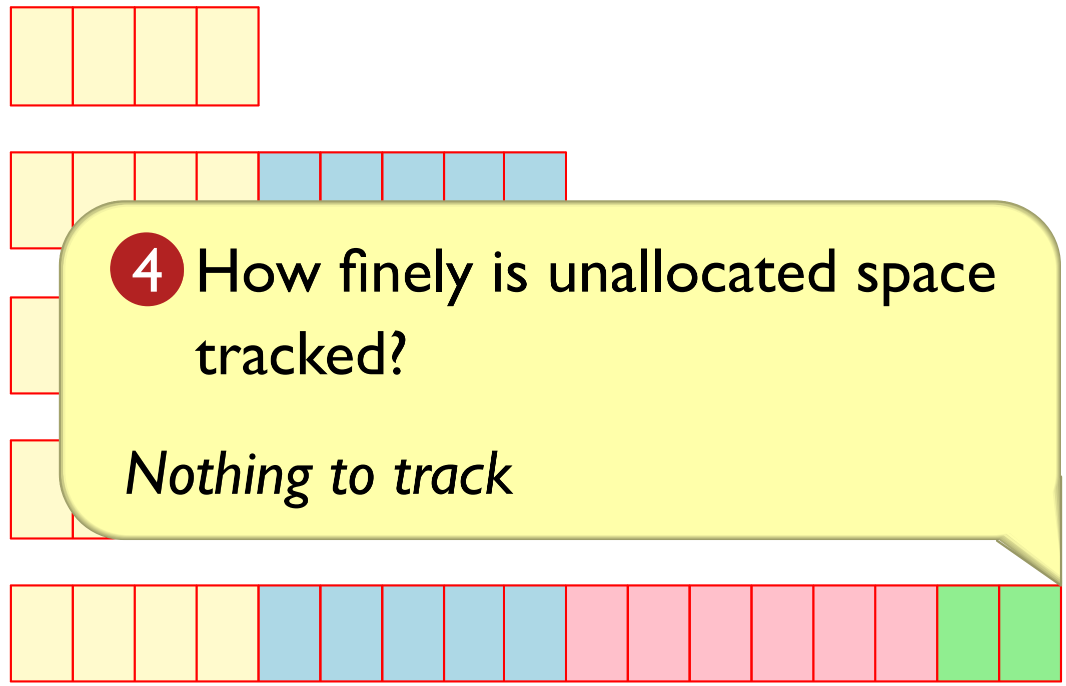
❸ How is unallocated space selected for each allocation?

Always add to the end



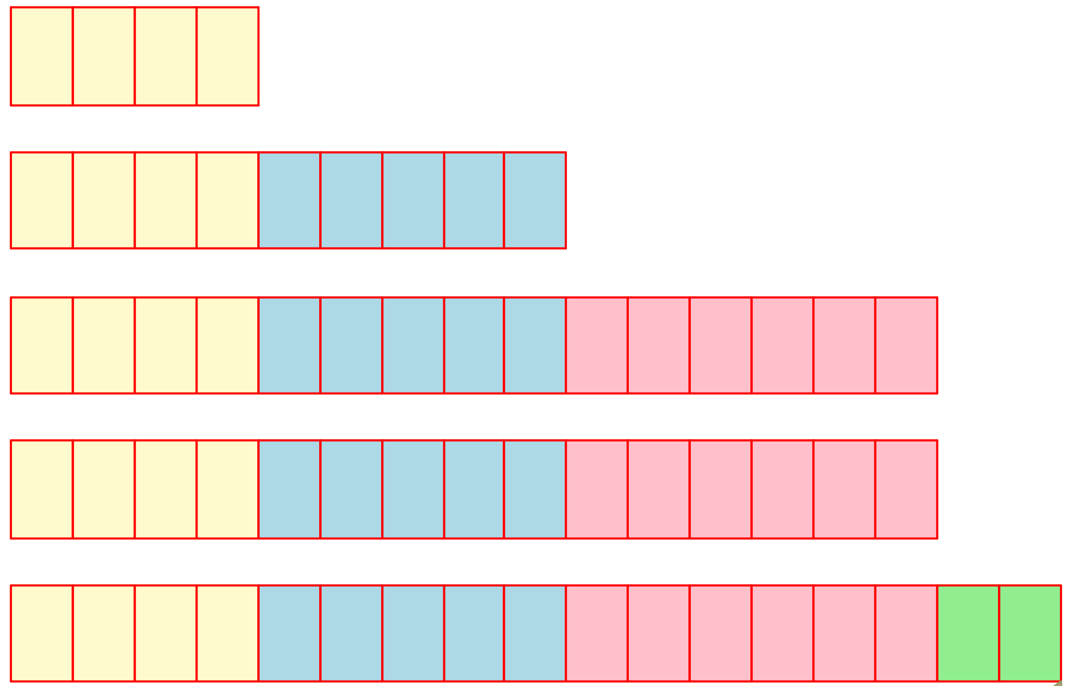
Naive `sbrk` Allocator

```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)
```



Naive `sbrk` Allocator

```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)
```



5 When are more pages needed from the kernel?

Every allocation

Naive `sbrk` Allocator

Real allocator needs to produce pointers aligned on 16 bytes:

```
#define ALIGNMENT 16
#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~(ALIGNMENT-1))
```

Copy

```
void *mm_malloc(size_t size) {
  return sbrk(ALIGN(size));
}

void mm_free(void *p) {
}
```

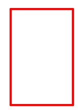
Copy

4 How finely is unallocated space tracked?

Some unallocated space can be left in a block for alignment padding

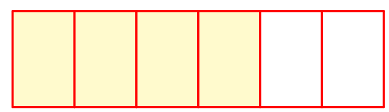
Picture Conventions

Since an implementation aligns to 16 bytes:

 = 16 bytes, a “word”

$\underline{N}$ = $N\times 16$ bytes

`p1 = malloc(4)`



allocation of 64 bytes

Naive Chunked `sbrk` Allocator

Chunk size of 6:

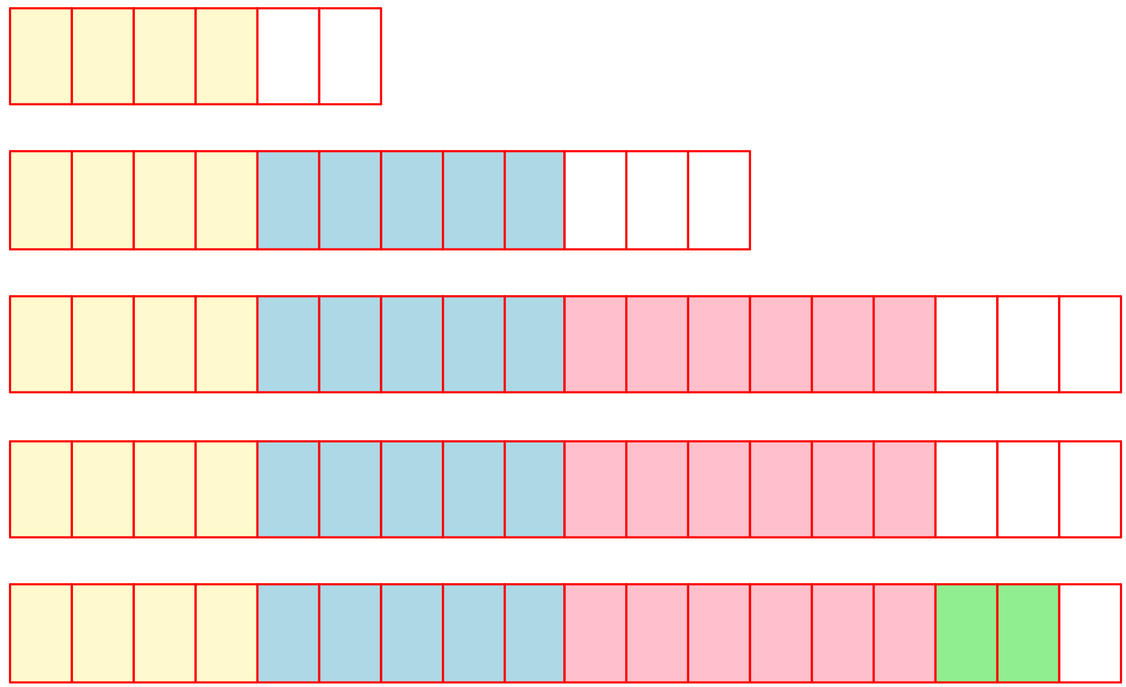
`p1 = malloc(4)`

`p2 = malloc(5)`

`p3 = malloc(6)`

`free(p2)`

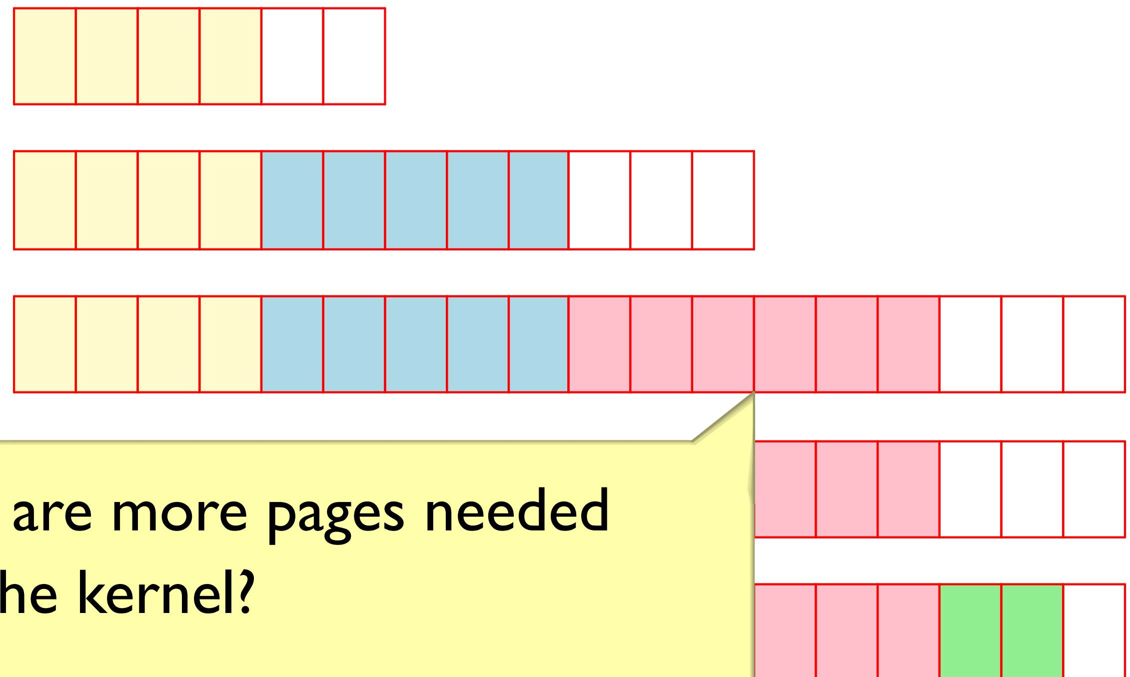
`p4 = malloc(2)`



Naive Chunked `sbrk` Allocator

Chunk size of 6:

```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = mall
```



5 When are more pages needed from the kernel?

When more is needed for an allocation

Naive Chunked `sbrk` Allocator

Pick a chunk size:

```
#define CHUNK_SIZE (1 << 14)
#define CHUNK_ALIGN(size) (((size)+(CHUNK_SIZE-1)) & ~(CHUNK_SIZE-1))
```

Copy

Naive Chunked `sbrk` Allocator

```
void *current_avail = NULL;
size_t current_avail_size = 0;

int mm_init() {
  current_avail = sbrk(0);
  current_avail_size = 0;
  return 0;
}
```

Copy

Naive Chunked `sbrk` Allocator

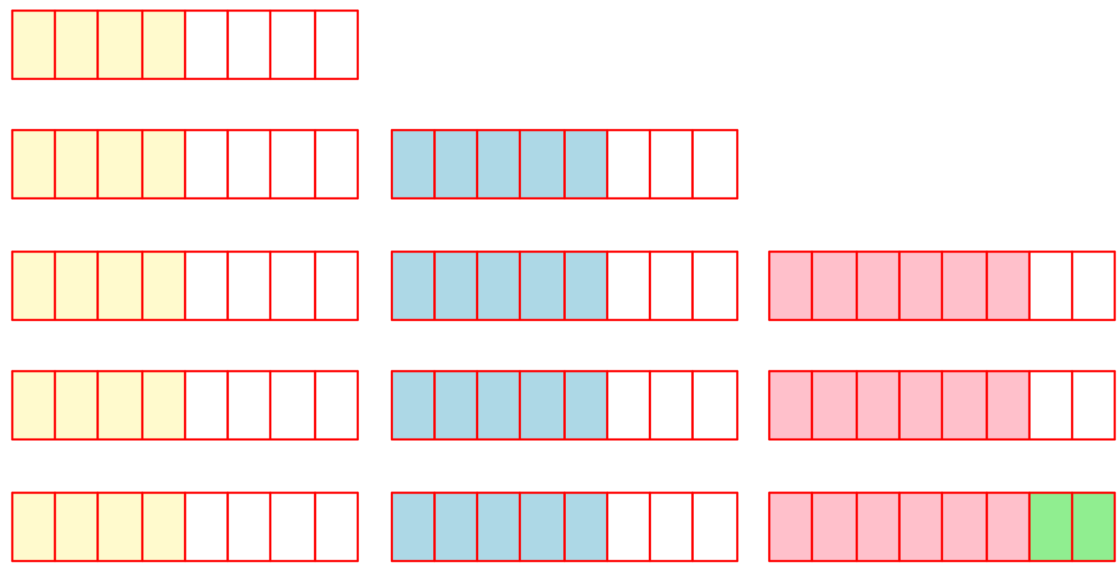
```
void *mm_malloc(size_t size) {
  size_t newsize = ALIGN(size);
  void *p;

  if (current_avail_size < newsize) {
    sbrk(CHUNK_ALIGN(newsize));
    current_avail_size += CHUNK_ALIGN(newsize);
  }

  p = current_avail;
  current_avail += newsize;
  current_avail_size -= newsize;

  return p;
}
```

Copy

Naive `mmap` Allocator

Page size of 8:

```
p1 = malloc(4)
p2 = malloc(5)
p3 = malloc(6)
free(p2)
p4 = malloc(2)
```



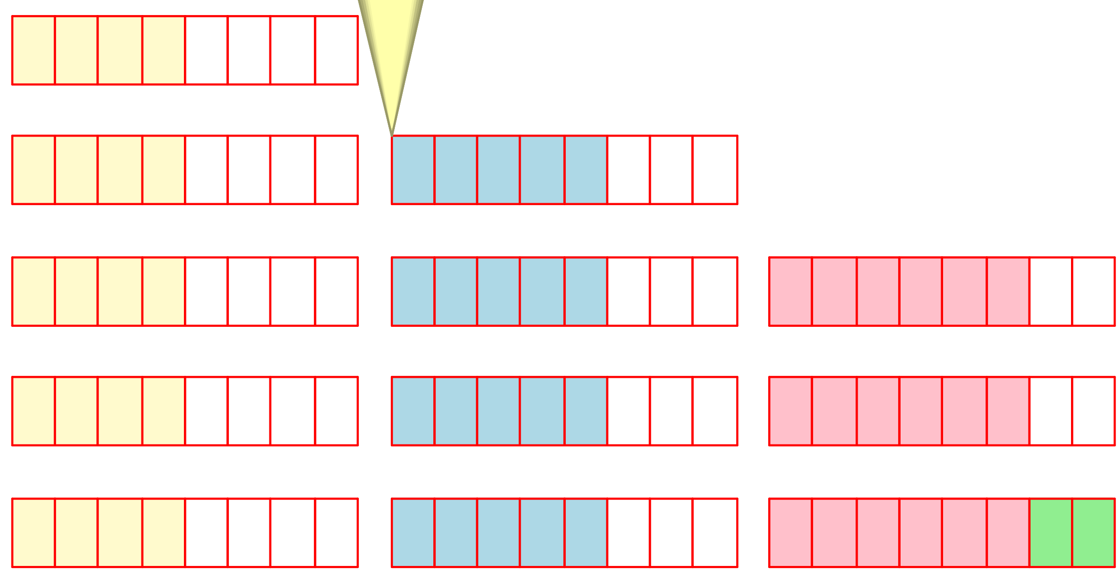
Naive `mmap` Allocator

5 When are more pages needed from the kernel?

When the most recent page doesn't have space

Page size of 8:

```
p1 = malloc(4)
p2 = malloc(5)
p3 = malloc(6)
free(p2)
p4 = malloc(2)
```



Naive `mmap` Allocator

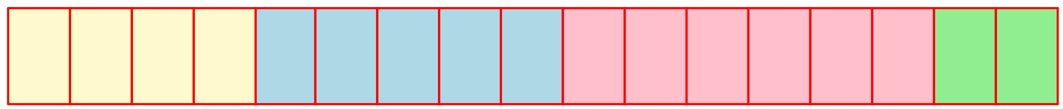
```
void *current_avail = NULL;
size_t current_avail_size = 0;

void *mm_malloc(size_t size) {
  size_t newsize = ALIGN(size);
  void *p;

  if (current_avail_size < newsize) {
    current_avail = mmap(0, CHUNK_ALIGN(newsize),
                         PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANON,
                         -1, 0);
    current_avail_size = CHUNK_ALIGN(newsize);
  }

  p = current_avail;
  current_avail += newsize;
  current_avail_size -= newsize;

  return p;
}
```

Copy

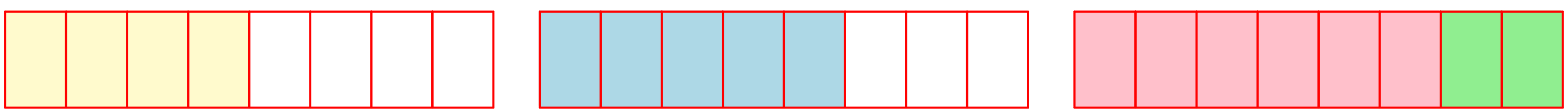
Fragmentation

Unallocated space in mapped pages is wasted

Naive **`sbrk`**:



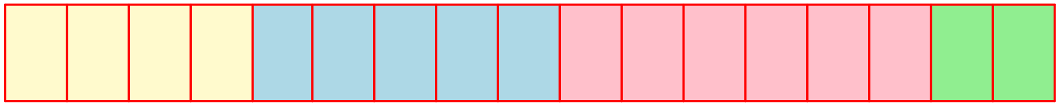
Naive **`mmap`**:



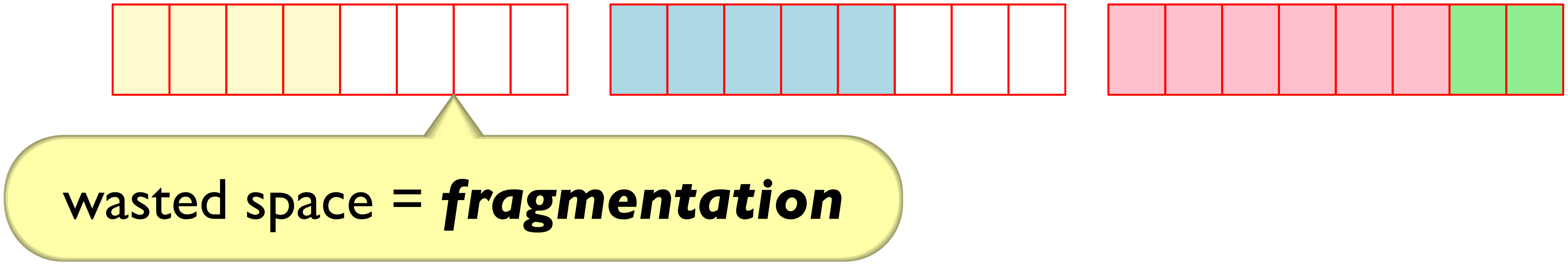
Fragmentation

Unallocated space in mapped pages is wasted

Naive **`sbrk`**:



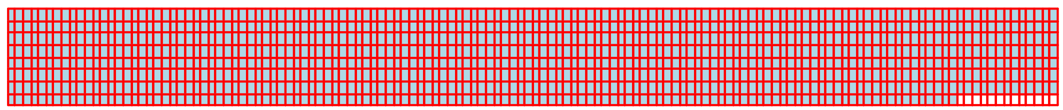
Naive **`mmap`**:



Fragmentation

Unallocated space in mapped pages is wasted

Naive **`sbrk`**:



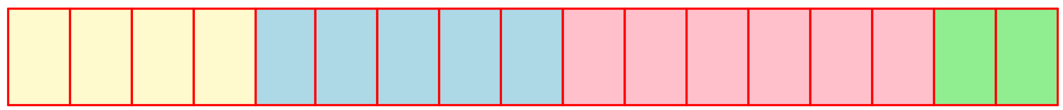
Naive **`mmap`**:

Pick page chunk ≫ allocation size

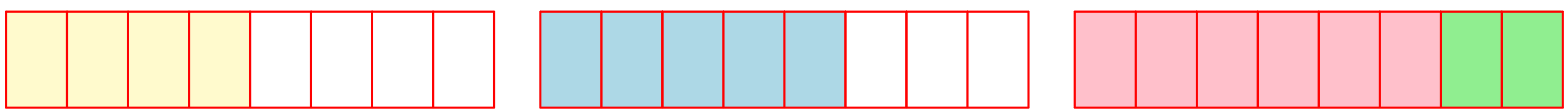
Fragmentation

Unallocated space in mapped pages is wasted

Naive **`sbrk`**:



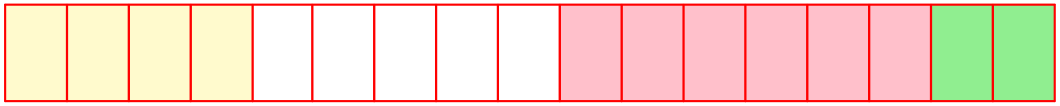
Naive **`mmap`**:



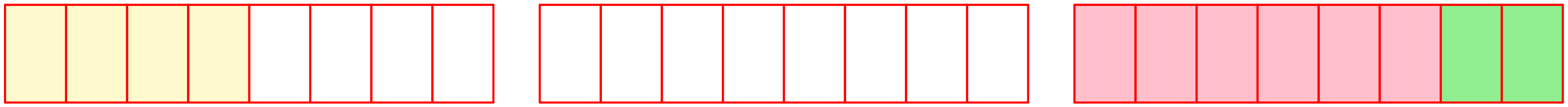
Fragmentation

Unallocated space in mapped pages is wasted

Naive **`sbrk`**:



Naive **`mmap`**:

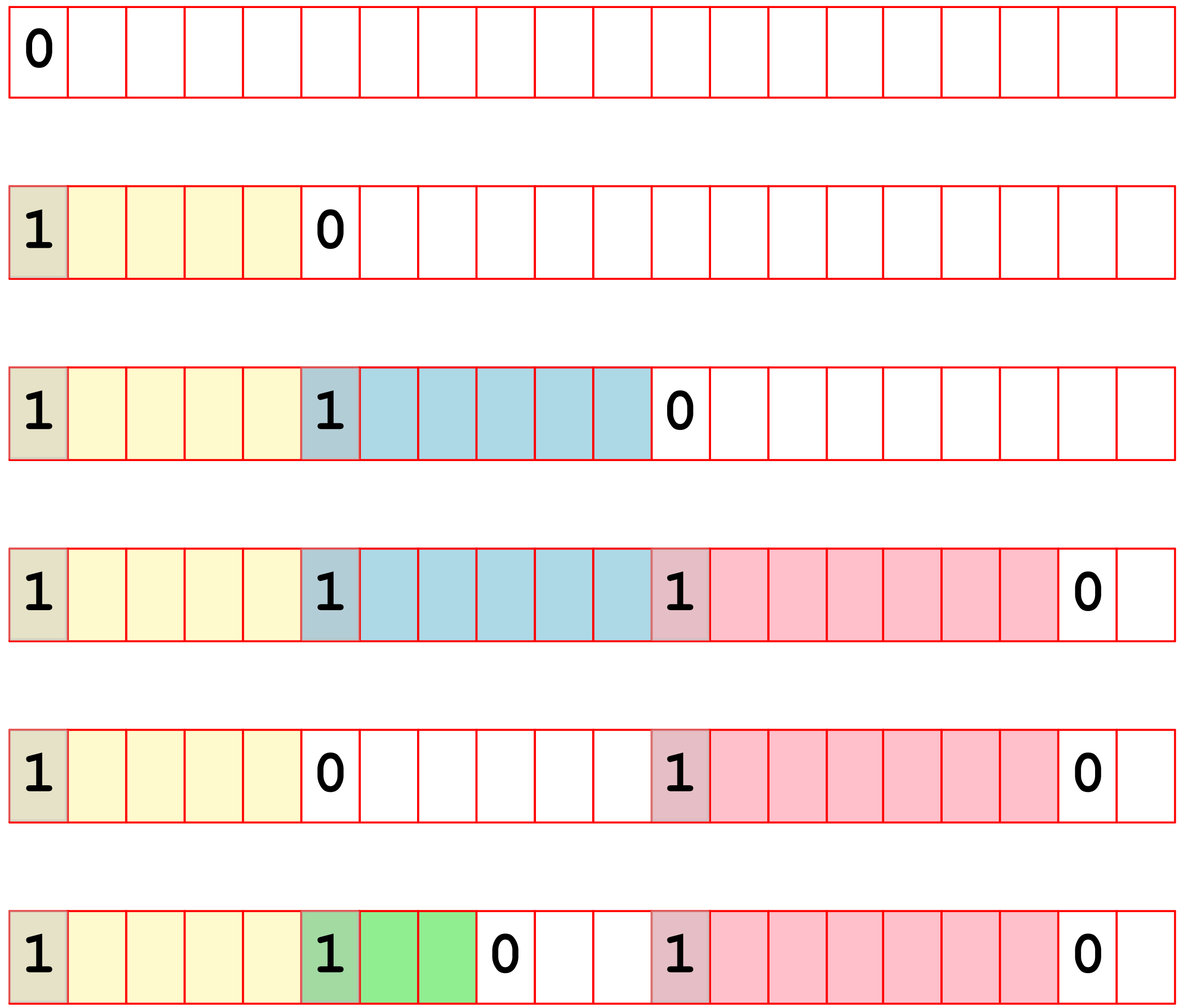


Taking **`free`** into account, both naive implementations suffer from extreme fragmentation

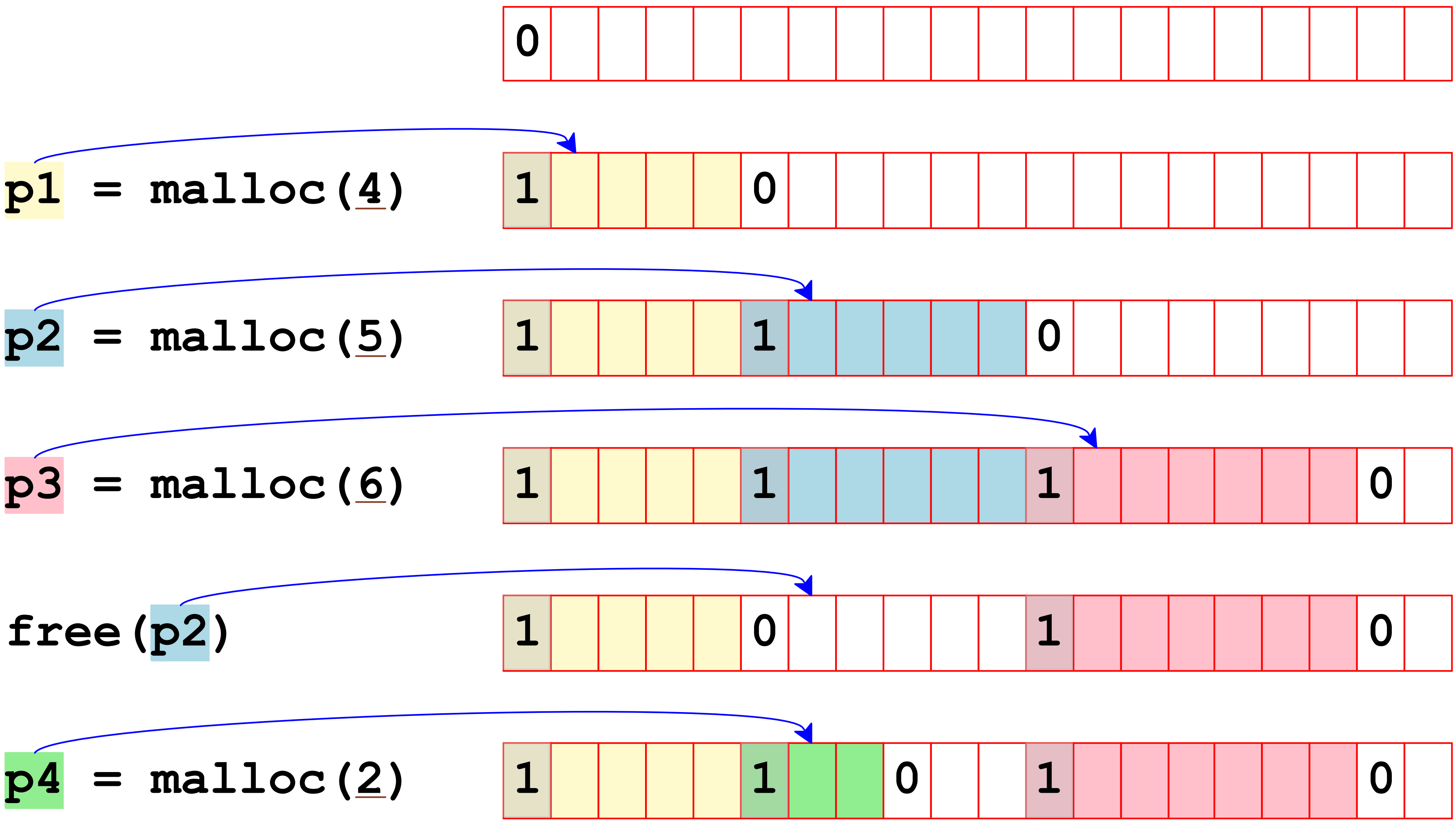
... so we need to keep track of unallocated space

Allocation Bit in a Block Header

```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)
```

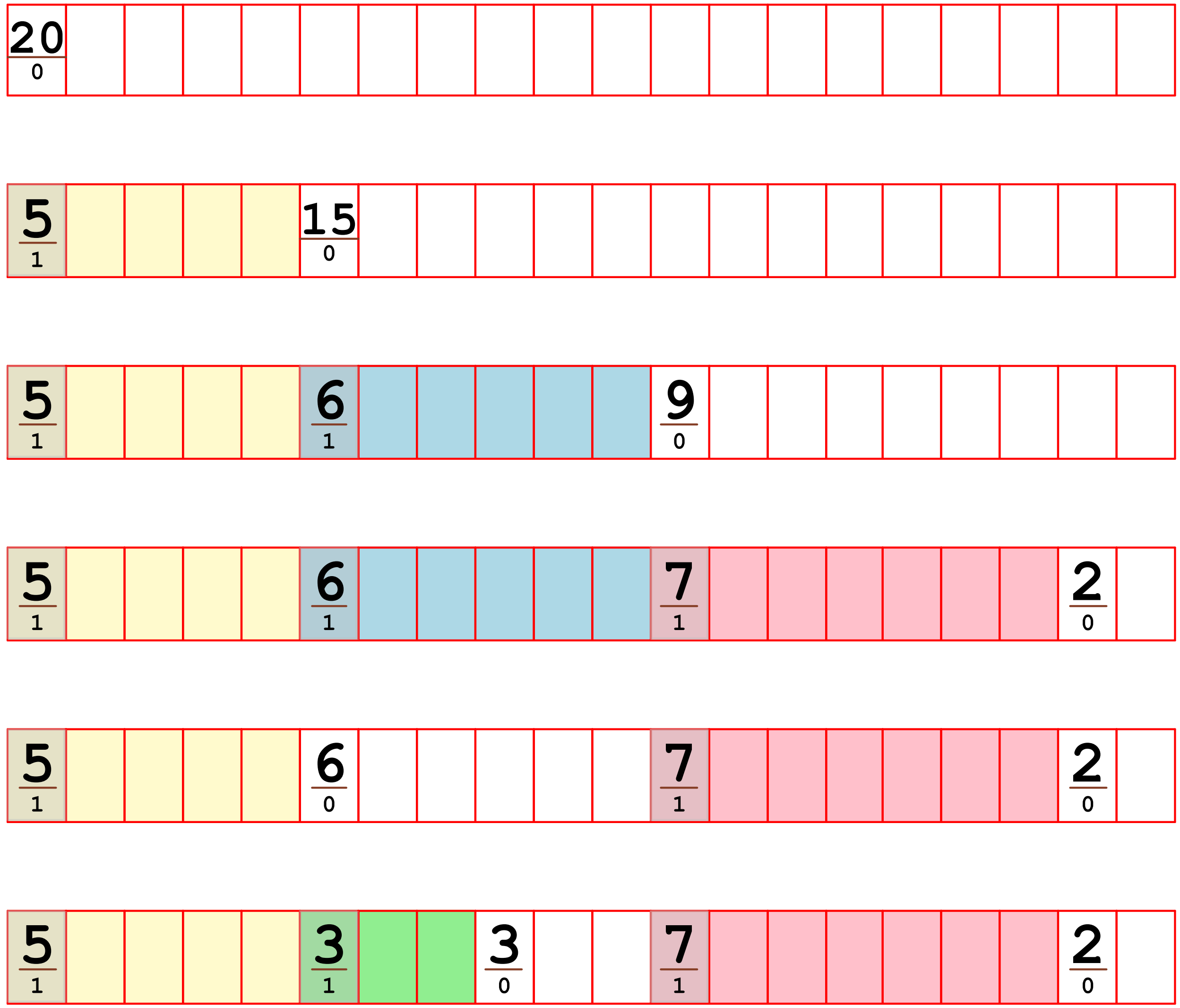


Allocation Bit in a Block Header



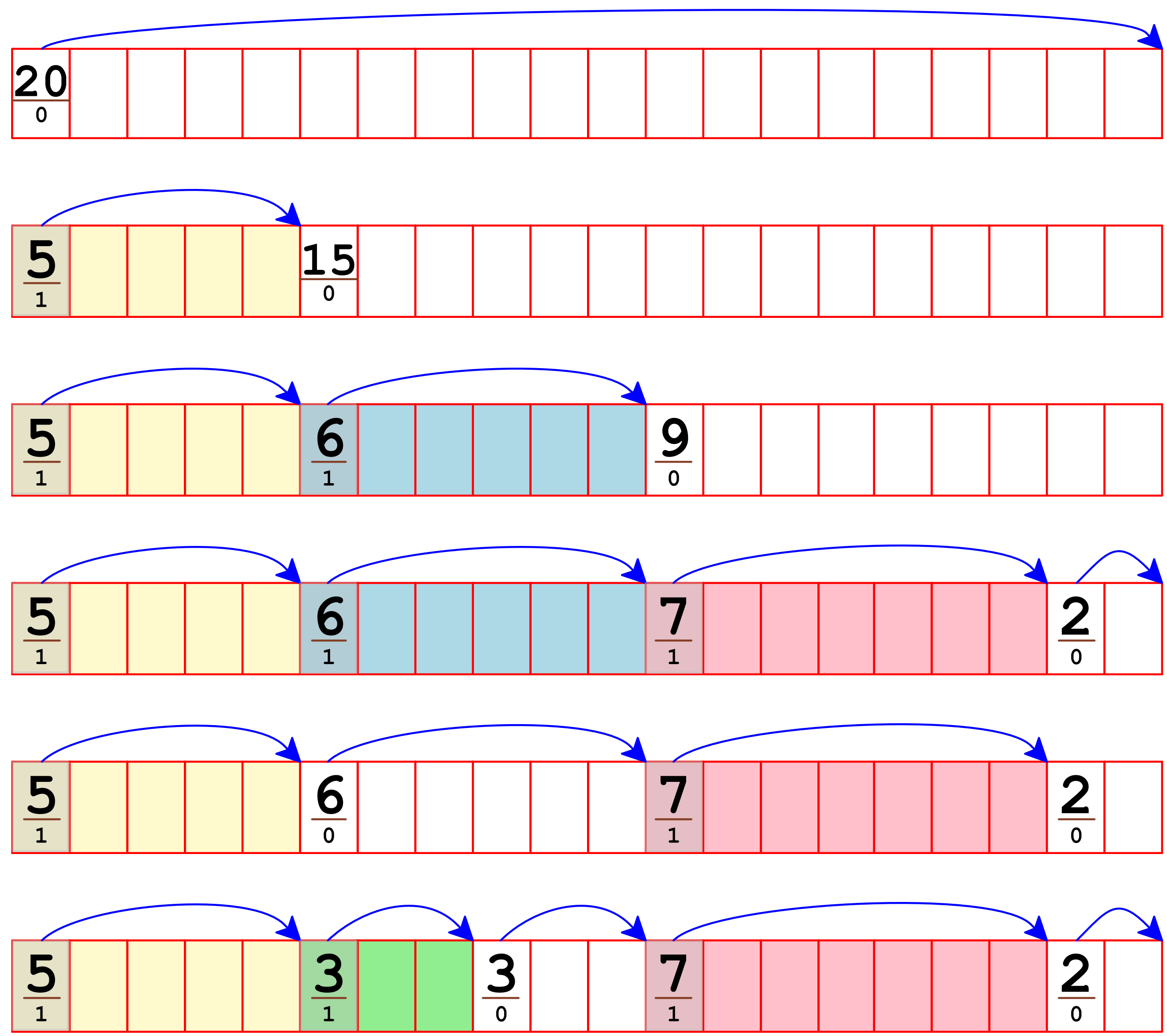
Size + Allocation Bit in a Block Header

```
p1 = malloc(4)
p2 = malloc(5)
p3 = malloc(6)
free(p2)
p4 = malloc(2)
```

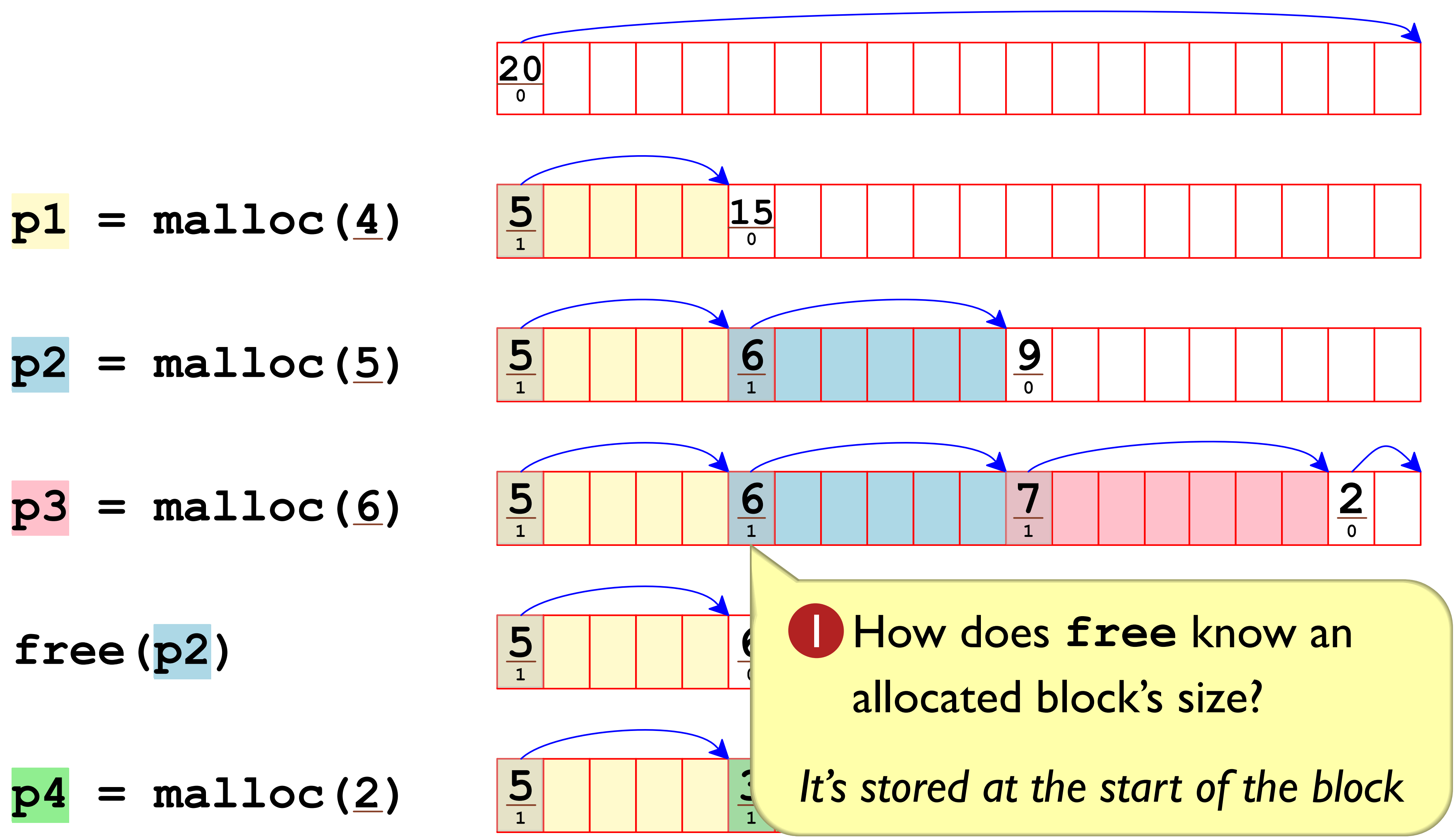


Sizes in a Block Header ⇒ Implicit Free List

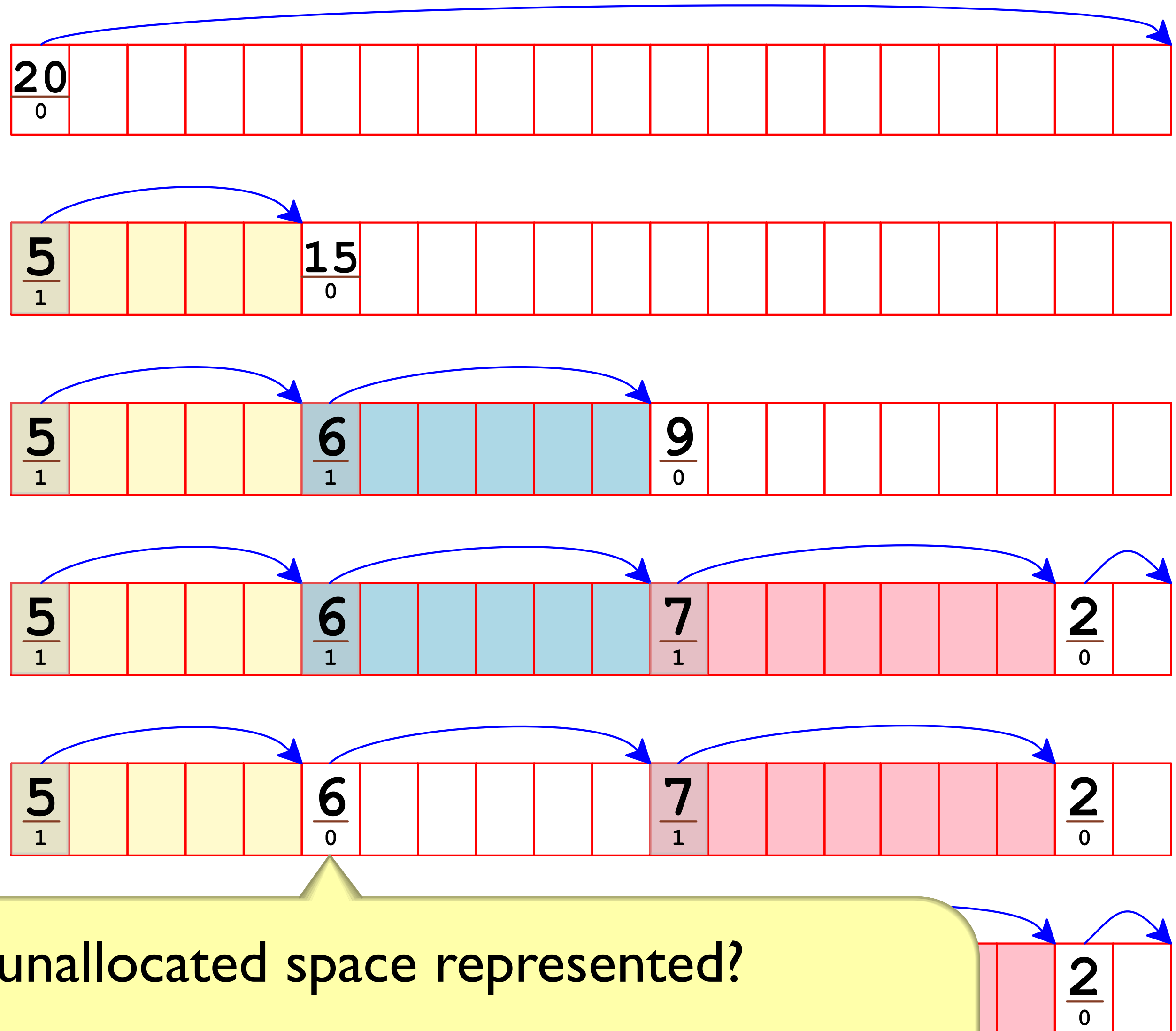
```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)
```



Sizes in a Block Header ⇒ Implicit Free List



Sizes in a Block Header ⇒ Implicit Free List



`p1 = malloc(4)`

`p2 = malloc(5)`

`p3 = malloc(6)`

`free(p2)`

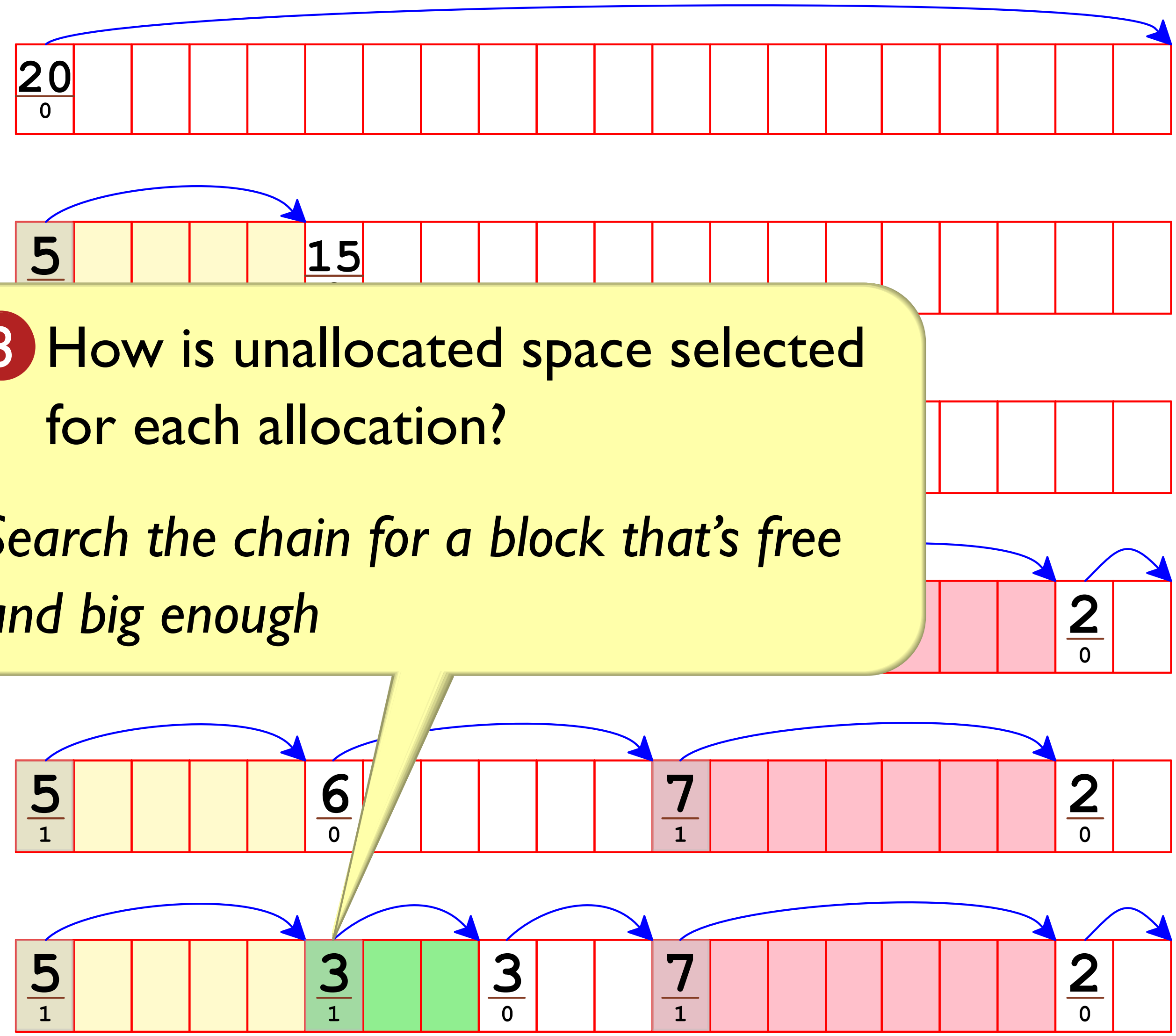
`p4 = mal`

2 How is unallocated space represented?

A bit in the block header distinguishes allocated from unallocated

Sizes in a Block Header ⇒ Implicit Free List

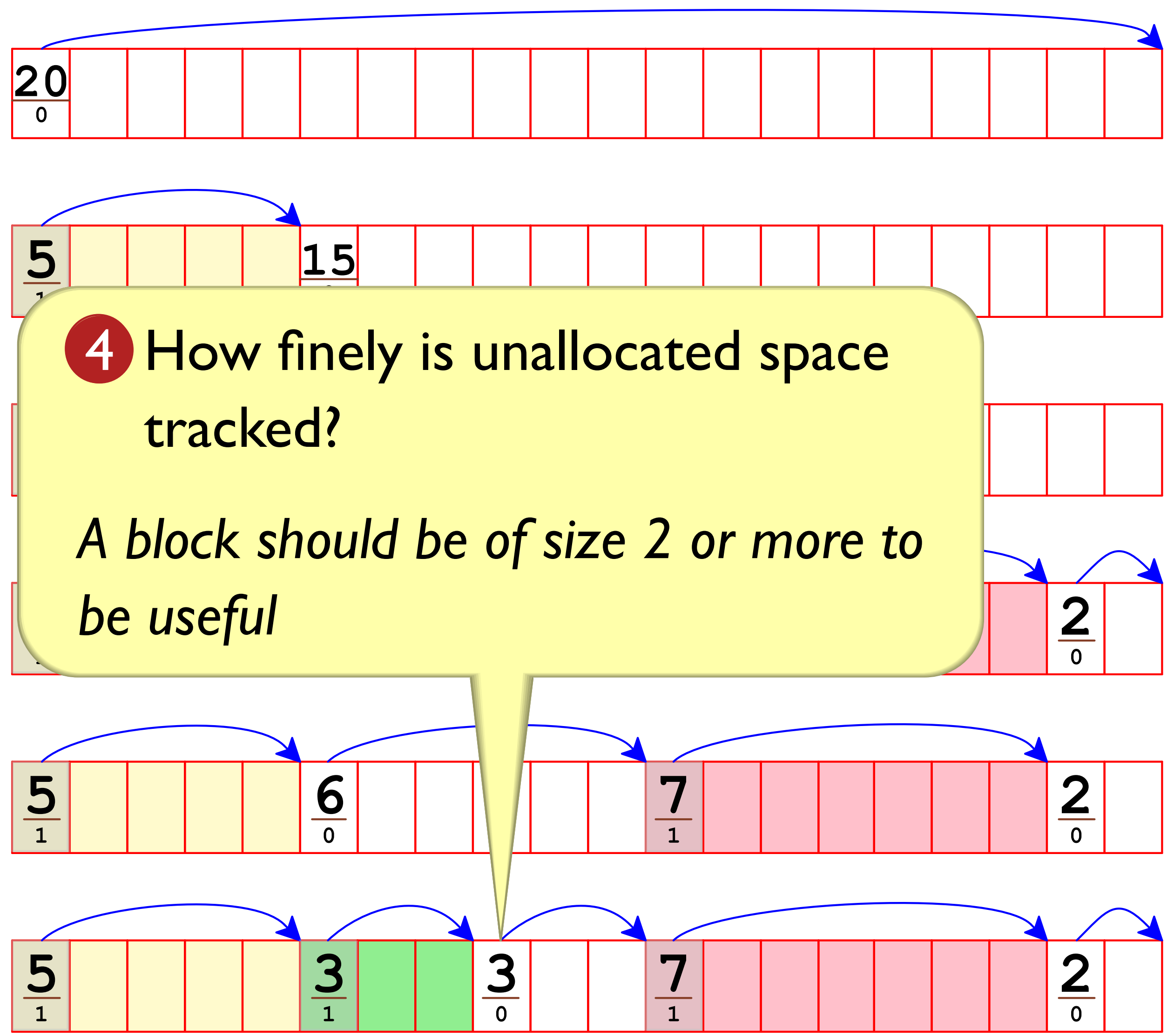
```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)
```



3 How is unallocated space selected for each allocation?

Search the chain for a block that's free and big enough

Sizes in a Block Header ⇒ Implicit Free List



`p1 = malloc(4)`

`p2 = malloc(5)`

`p3 = malloc(6)`

`free(p2)`

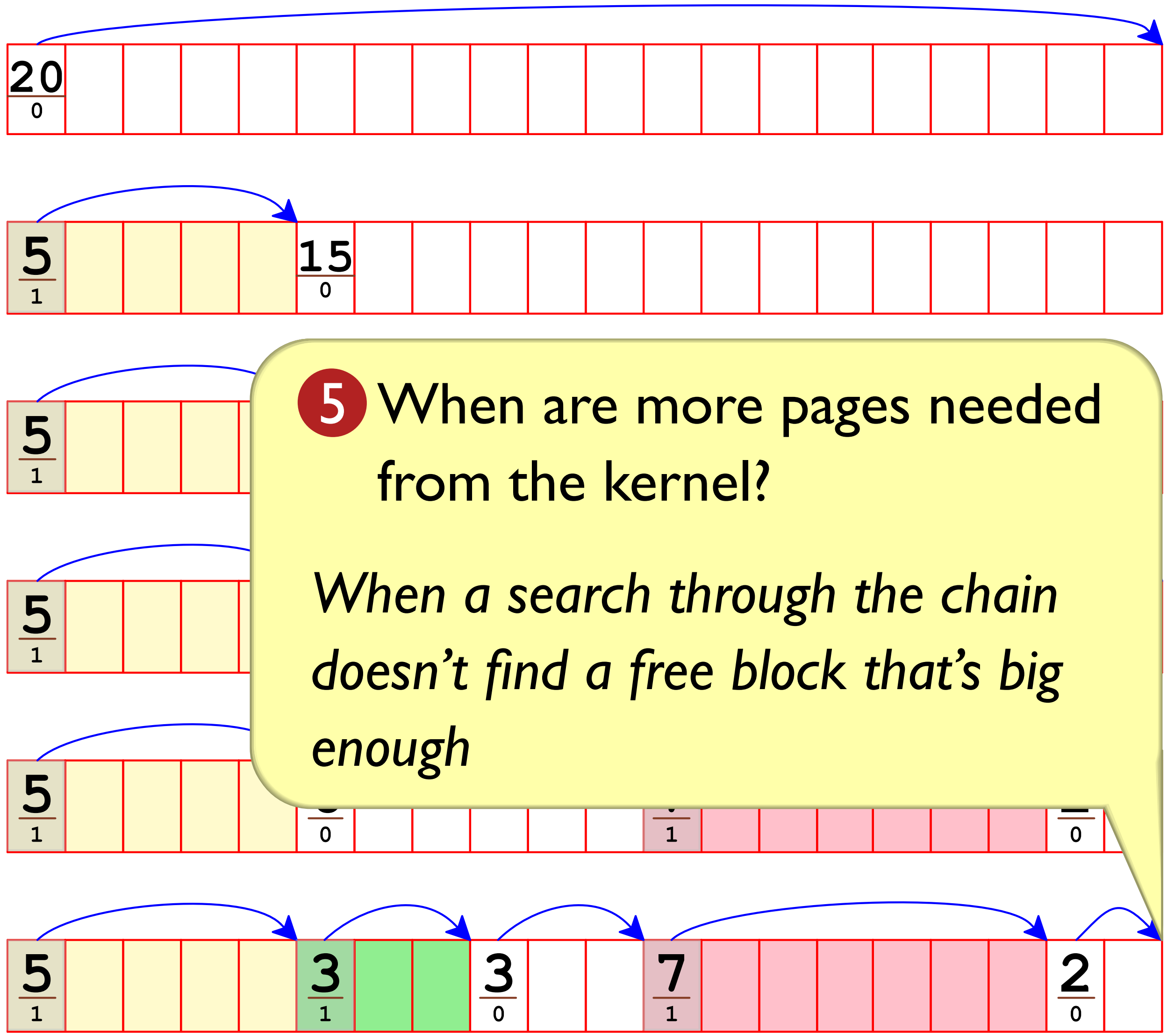
`p4 = malloc(2)`

4 How finely is unallocated space tracked?

A block should be of size 2 or more to be useful

Sizes in a Block Header ⇒ Implicit Free List

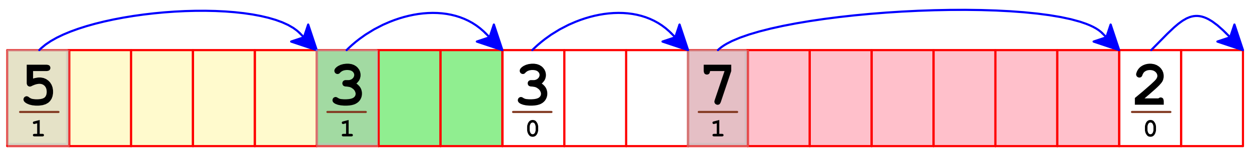
```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)
```



5 When are more pages needed from the kernel?

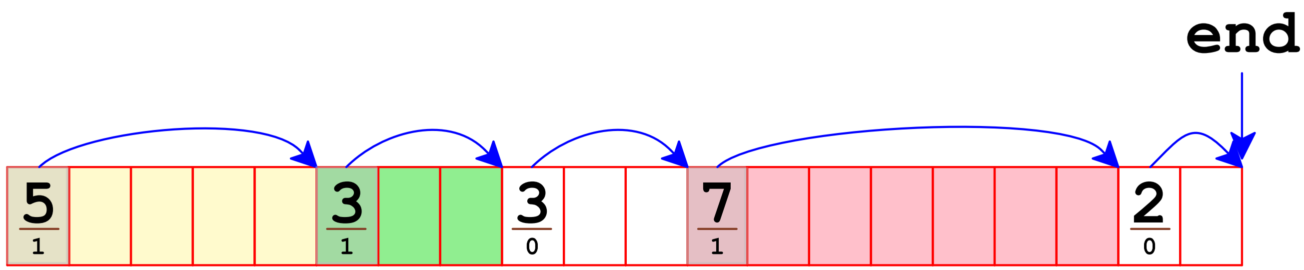
When a search through the chain doesn't find a free block that's big enough

Terminating the Block List

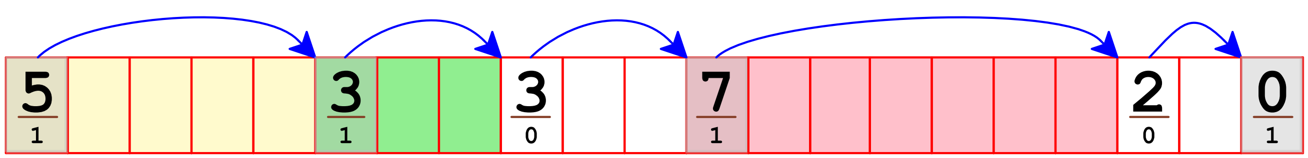


How does the allocator know that the size-2 block is the last one?

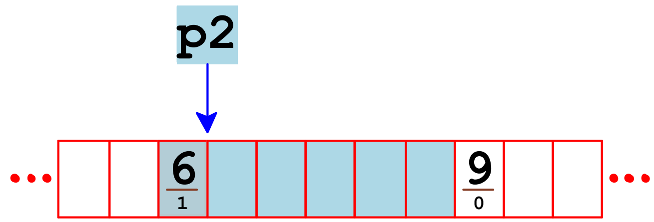
Compare the next pointer to an end-of-heap address



or Add a “zero”-sized block to terminate the chain



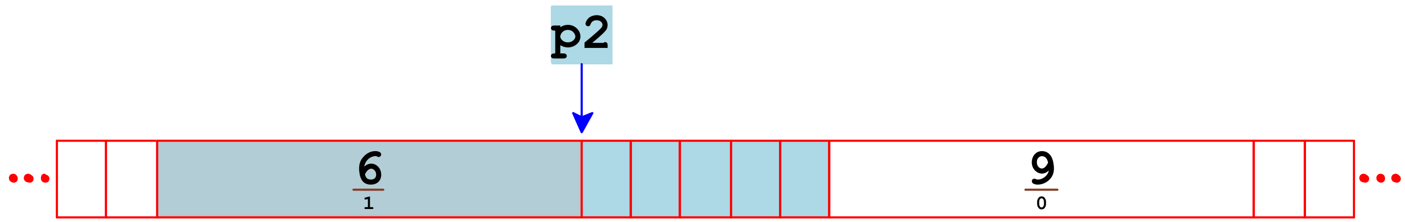
Storing the Size and Allocation Bit




```
typedef struct {
  size_t size;
  char   allocated;
} block_header;
```

Copy

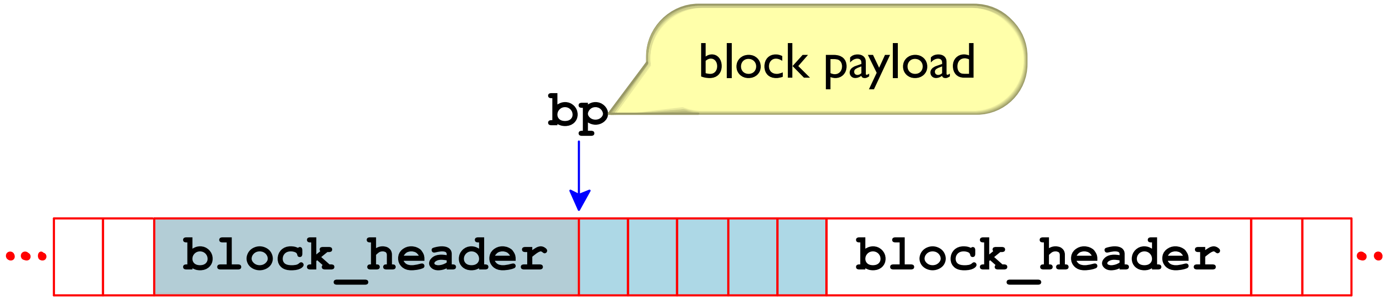
Storing the Size and Allocation Bit




```
typedef struct {
  size_t size;
  char   allocated;
} block_header;
```

Copy

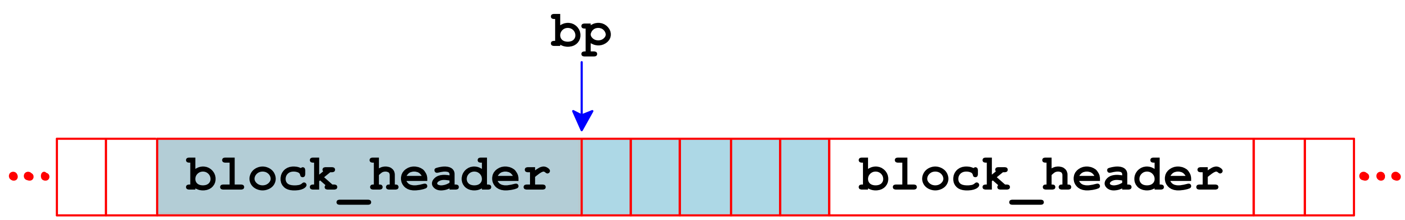
Storing the Size and Allocation Bit




```
typedef struct {
  size_t size;
  char   allocated;
} block_header;
```

Copy

Storing the Size and Allocation Bit




```
typedef struct {
  size_t size;
  char   allocated;
} block_header;
```

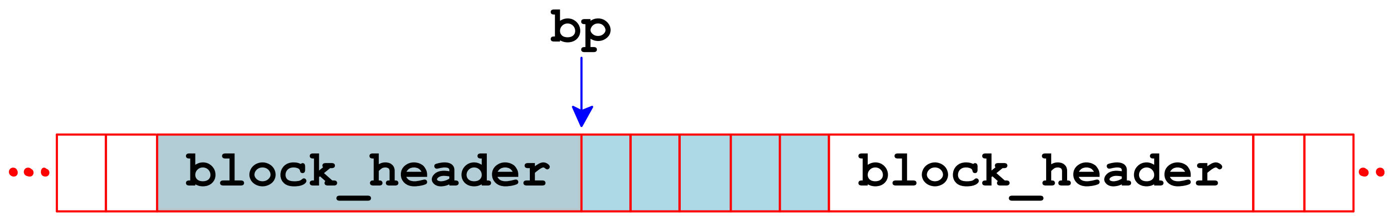
Copy

`sizeof(block_header) = 16`

Aligned payload size ⇒ 16-byte alignment preserved

... although that's a lot of empty space

Storing the Size and Allocation Bit




```
typedef struct {
  size_t size;
  char   allocated;
} block_header;
```

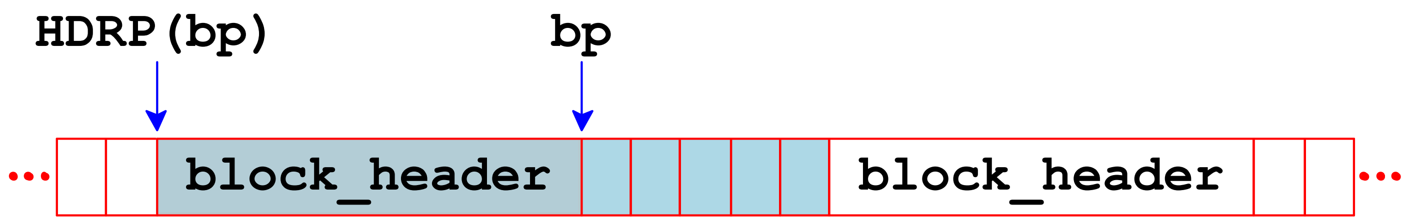
Copy

Macro for block overhead:

```
#define OVERHEAD sizeof(block_header)
```

Copy

Storing the Size and Allocation Bit




```
typedef struct {
  size_t size;
  char   allocated;
} block_header;
```

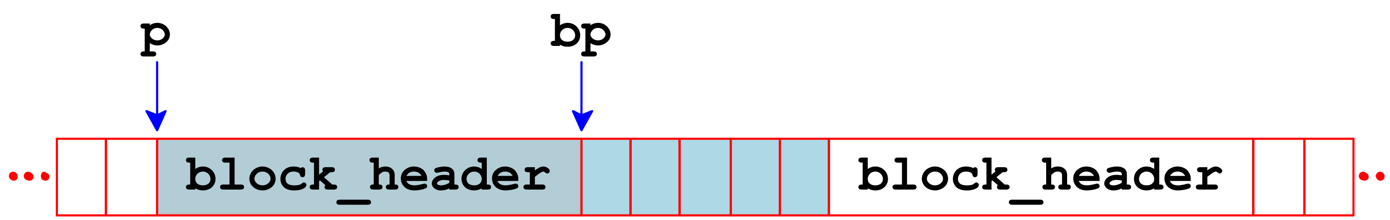
Copy

Macro for getting the header from a payload pointer:

```
#define HDRP(bp) ((char *)(bp) - sizeof(block_header))
```

Copy

Storing the Size and Allocation Bit




```
typedef struct {
  size_t size;
  char   allocated;
} block_header;
```

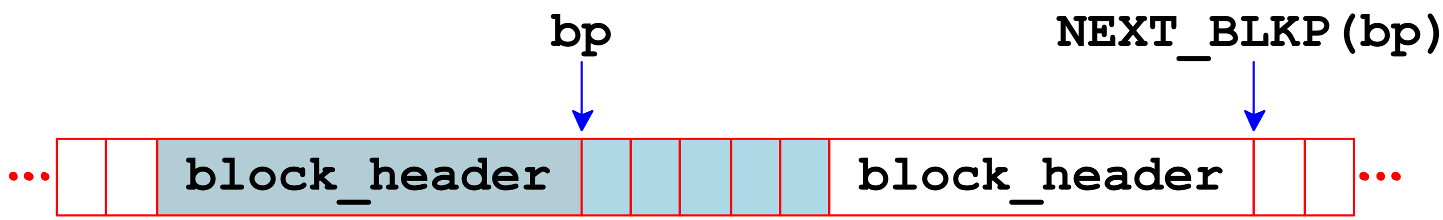
Copy

Macros for working with a raw pointer as the header:

```
#define GET_SIZE(p)   ((block_header *)(p))->size
#define GET_ALLOC(p)  ((block_header *)(p))->allocated
```

Copy

Storing the Size and Allocation Bit




```
typedef struct {
  size_t size;
  char   allocated;
} block_header;
```

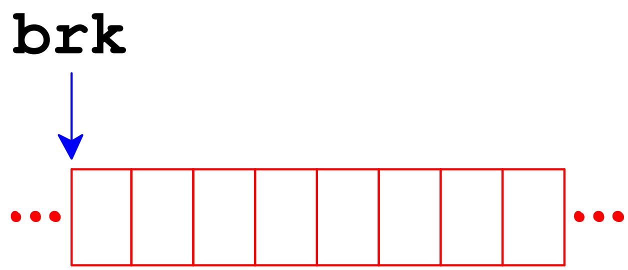
Copy

Macro for getting the next block's payload:

```
#define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)))
```

Copy

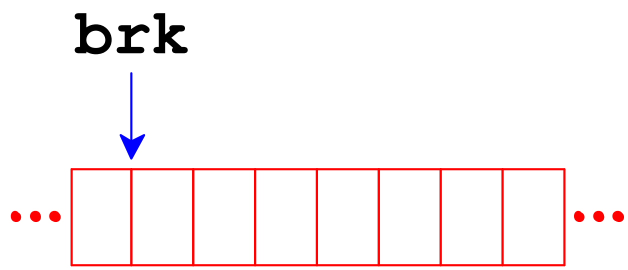
Initializing the Allocator




```
void *first_bp;

int mm_init() {
  sbrk(sizeof(block_header));
  first_bp = sbrk(0);

  GET_SIZE(HDRP(first_bp)) = 0;
  GET_ALLOC(HDRP(first_bp)) = 1;

  return 0;
}
```

Copy

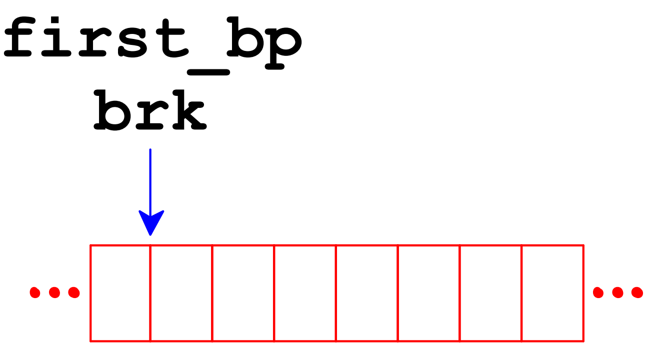
Initializing the Allocator




```
void *first_bp;

int mm_init() {
  sbrk(sizeof(block_header));
  first_bp = sbrk(0);

  GET_SIZE(HDRP(first_bp)) = 0;
  GET_ALLOC(HDRP(first_bp)) = 1;

  return 0;
}
```

Copy

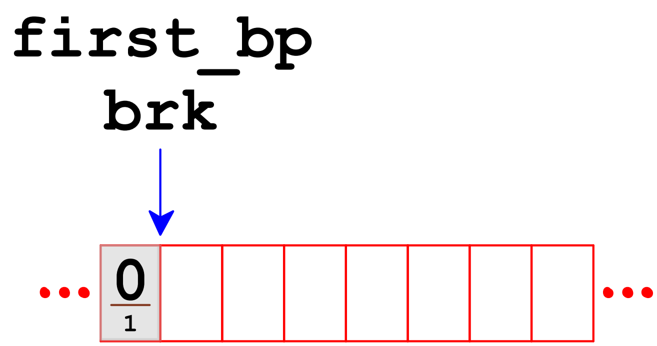
Initializing the Allocator




```
void *first_bp;

int mm_init() {
  sbrk(sizeof(block_header));
  first_bp = sbrk(0);

  GET_SIZE(HDRP(first_bp)) = 0;
  GET_ALLOC(HDRP(first_bp)) = 1;

  return 0;
}
```

Copy

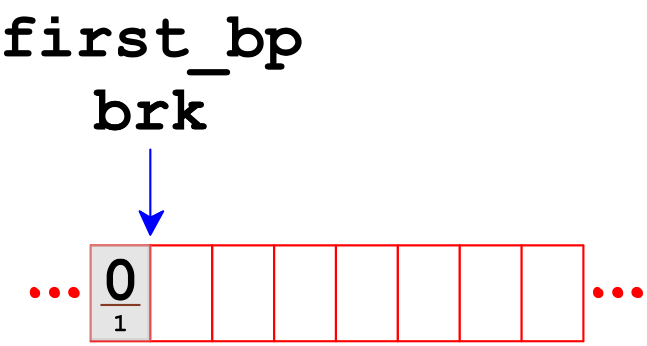
Initializing the Allocator




```
void *first_bp;

int mm_init() {
  sbrk(sizeof(block_header));
  first_bp = sbrk(0);

  GET_SIZE(HDRP(first_bp)) = 0;
  GET_ALLOC(HDRP(first_bp)) = 1;

  return 0;
}
```

Copy

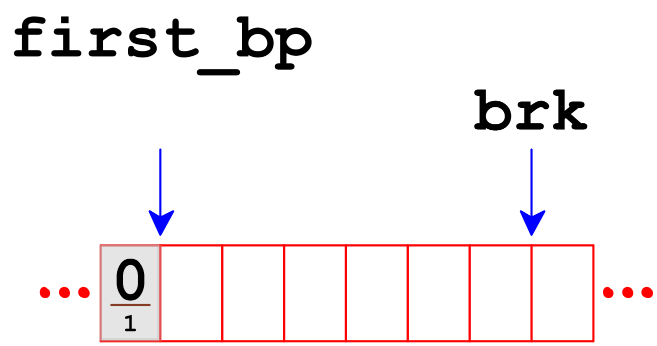
Adding Pages




```
void extend(size_t new_size) {
  size_t chunk_size = CHUNK_ALIGN(new_size);
  void *bp = sbrk(chunk_size);

  GET_SIZE(HDRP(bp)) = chunk_size;
  GET_ALLOC(HDRP(bp)) = 0;

  GET_SIZE(HDRP(NEXT_BLKP(bp))) = 0;
  GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 1;
}
```

Copy

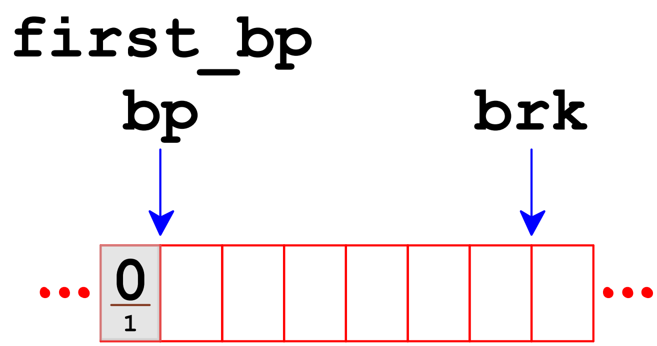
Adding Pages




```
void extend(size_t new_size) {
  size_t chunk_size = CHUNK_ALIGN(new_size);
  void *bp = sbrk(chunk_size);

  GET_SIZE(HDRP(bp)) = chunk_size;
  GET_ALLOC(HDRP(bp)) = 0;

  GET_SIZE(HDRP(NEXT_BLKP(bp))) = 0;
  GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 1;
}
```

Copy

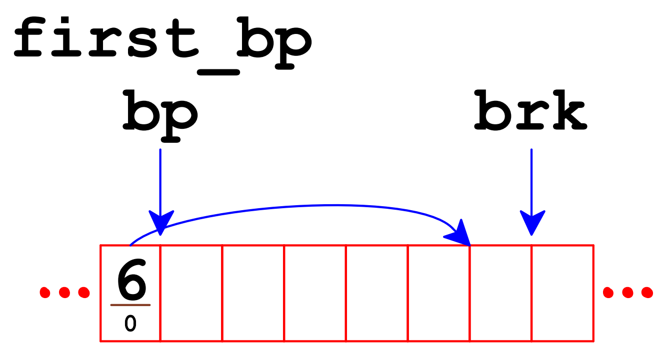
Adding Pages




```
void extend(size_t new_size) {
  size_t chunk_size = CHUNK_ALIGN(new_size);
  void *bp = sbrk(chunk_size);

  GET_SIZE(HDRP(bp)) = chunk_size;
  GET_ALLOC(HDRP(bp)) = 0;

  GET_SIZE(HDRP(NEXT_BLKP(bp))) = 0;
  GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 1;
}
```

Copy

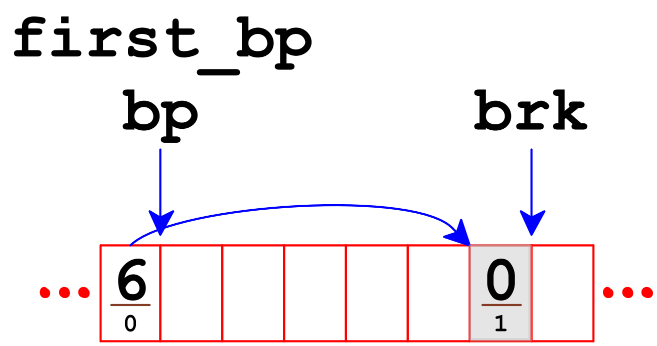
Adding Pages




```
void extend(size_t new_size) {
  size_t chunk_size = CHUNK_ALIGN(new_size);
  void *bp = sbrk(chunk_size);

  GET_SIZE(HDRP(bp)) = chunk_size;
  GET_ALLOC(HDRP(bp)) = 0;

  GET_SIZE(HDRP(NEXT_BLKP(bp))) = 0;
  GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 1;
}
```

Copy

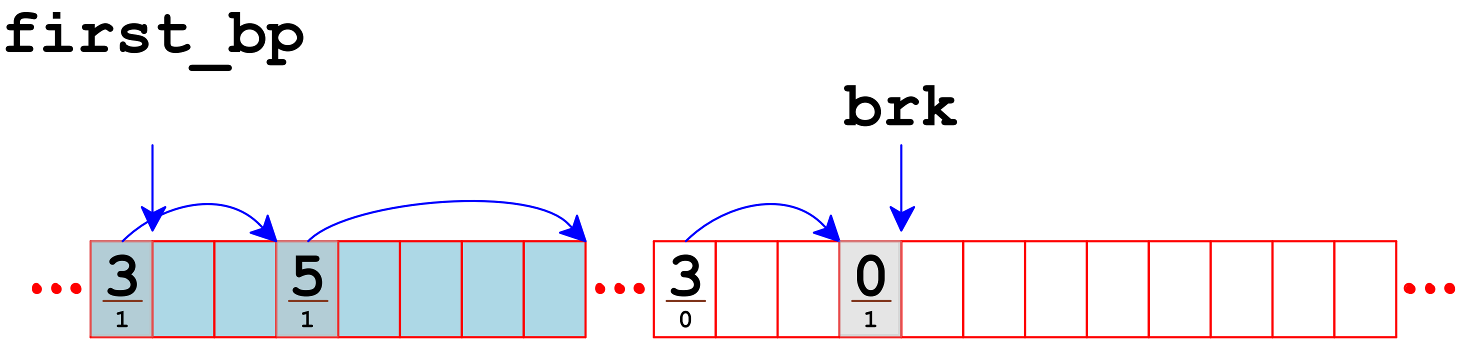
Adding Pages




```
void extend(size_t new_size) {
  size_t chunk_size = CHUNK_ALIGN(new_size);
  void *bp = sbrk(chunk_size);

  GET_SIZE(HDRP(bp)) = chunk_size;
  GET_ALLOC(HDRP(bp)) = 0;

  GET_SIZE(HDRP(NEXT_BLKP(bp))) = 0;
  GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 1;
}
```

Copy

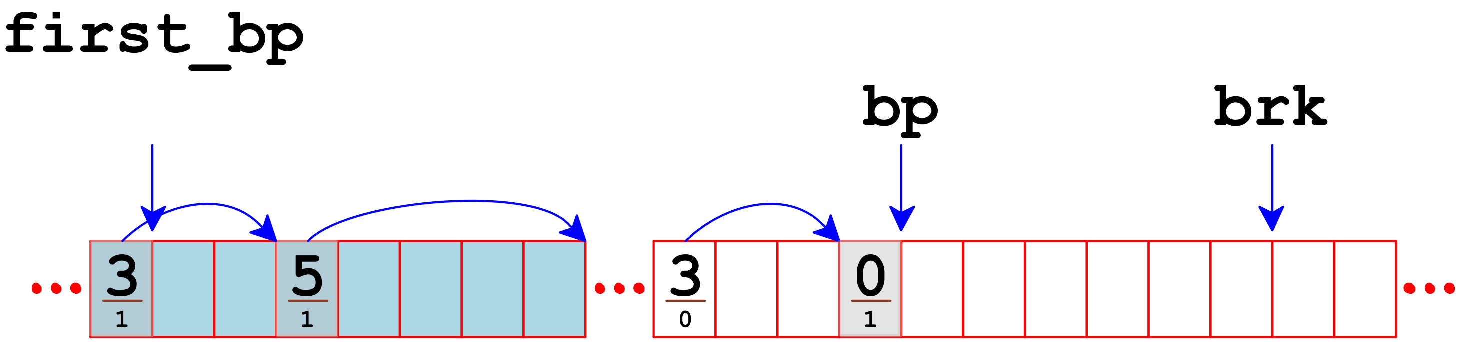
Adding Pages




```
void extend(size_t new_size) {
  size_t chunk_size = CHUNK_ALIGN(new_size);
  void *bp = sbrk(chunk_size);

  GET_SIZE(HDRP(bp)) = chunk_size;
  GET_ALLOC(HDRP(bp)) = 0;

  GET_SIZE(HDRP(NEXT_BLKP(bp))) = 0;
  GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 1;
}
```

Copy

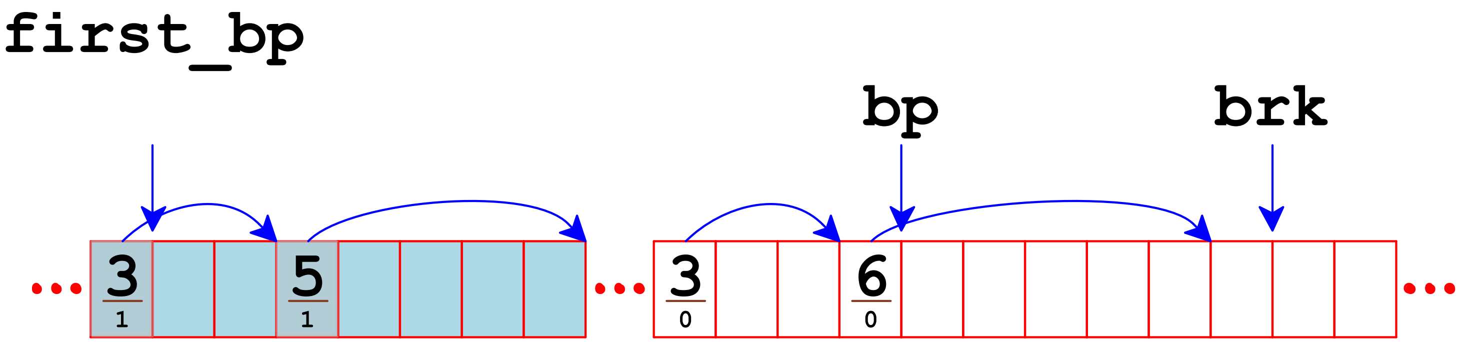
Adding Pages




```
void extend(size_t new_size) {
  size_t chunk_size = CHUNK_ALIGN(new_size);
  void *bp = sbrk(chunk_size);

  GET_SIZE(HDRP(bp)) = chunk_size;
  GET_ALLOC(HDRP(bp)) = 0;

  GET_SIZE(HDRP(NEXT_BLKP(bp))) = 0;
  GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 1;
}
```

Copy

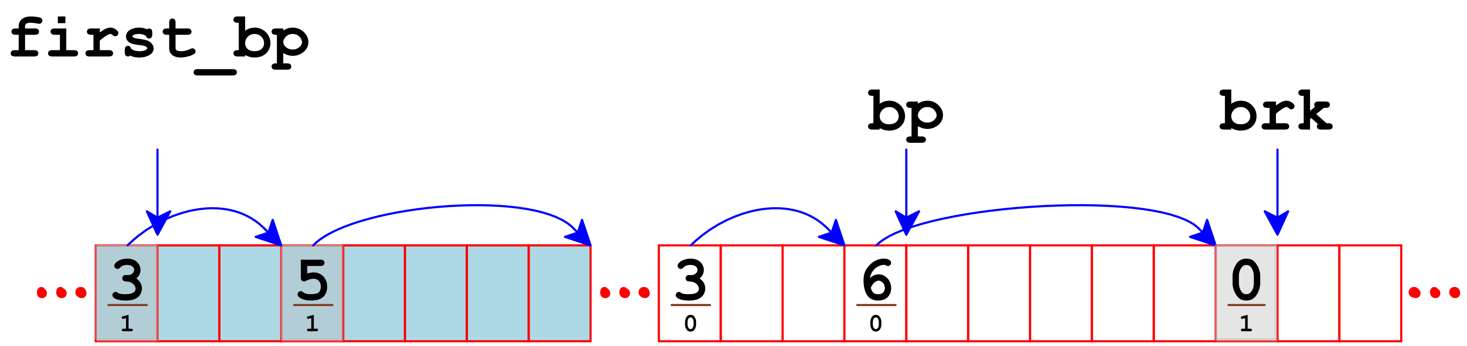
Adding Pages




```
void extend(size_t new_size) {
  size_t chunk_size = CHUNK_ALIGN(new_size);
  void *bp = sbrk(chunk_size);

  GET_SIZE(HDRP(bp)) = chunk_size;
  GET_ALLOC(HDRP(bp)) = 0;

  GET_SIZE(HDRP(NEXT_BLKP(bp))) = 0;
  GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 1;
}
```

Copy

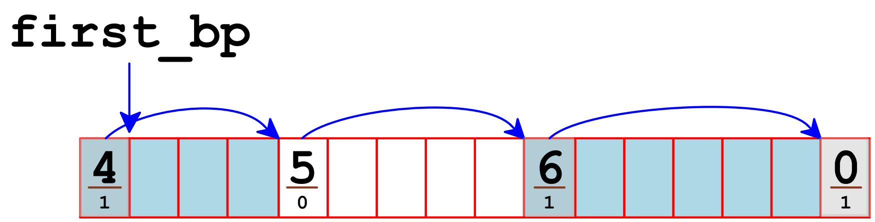
Adding Pages




```
void extend(size_t new_size) {
  size_t chunk_size = CHUNK_ALIGN(new_size);
  void *bp = sbrk(chunk_size);

  GET_SIZE(HDRP(bp)) = chunk_size;
  GET_ALLOC(HDRP(bp)) = 0;

  GET_SIZE(HDRP(NEXT_BLKP(bp))) = 0;
  GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 1;
}
```

Copy

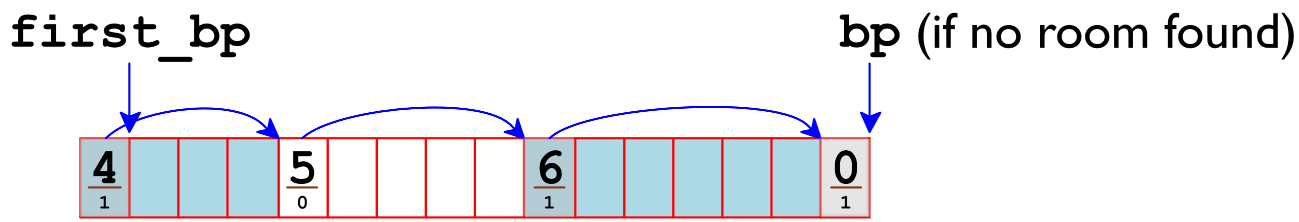
Finding a Block to Allocate




```
void *mm_malloc(size_t size) {
  int new_size = ALIGN(size + OVERHEAD);
  void *bp = first_bp;

  while (GET_SIZE(HDRP(bp)) != 0) {
    if (!GET_ALLOC(HDRP(bp))
        && (GET_SIZE(HDRP(bp)) >= new_size)) {
      set_allocated(bp, new_size);
      return bp;
    }
    bp = NEXT_BLKP(bp);
  }

  extend(new_size);
  set_allocated(bp, new_size);
  return bp;
}
```

Copy

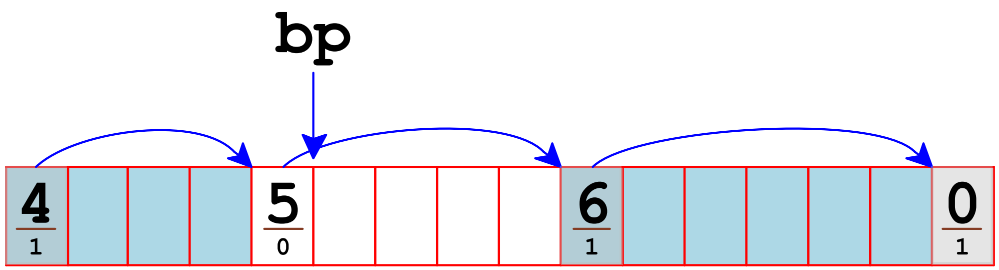
Finding a Block to Allocate




```
void *mm_malloc(size_t size) {
  int new_size = ALIGN(size + OVERHEAD);
  void *bp = first_bp;

  while (GET_SIZE(HDRP(bp)) != 0) {
    if (!GET_ALLOC(HDRP(bp))
        && (GET_SIZE(HDRP(bp)) >= new_size)) {
      set_allocated(bp, new_size);
      return bp;
    }
    bp = NEXT_BLKP(bp);
  }

  extend(new_size);
  set_allocated(bp, new_size);
  return bp;
}
```

Copy

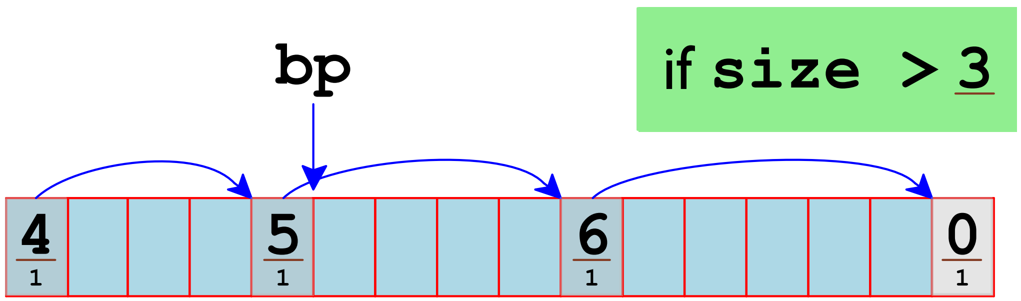
Marking a Block as Allocated




```
void set_allocated(void *bp, size_t size) {
  size_t extra_size = GET_SIZE(HDRP(bp)) - size;

  if (extra_size > ALIGN(1 + OVERHEAD)) {
    GET_SIZE(HDRP(bp)) = size;
    GET_SIZE(HDRP(NEXT_BLKP(bp))) = extra_size;
    GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 0;
  }

  GET_ALLOC(HDRP(bp)) = 1;
}
```

Copy

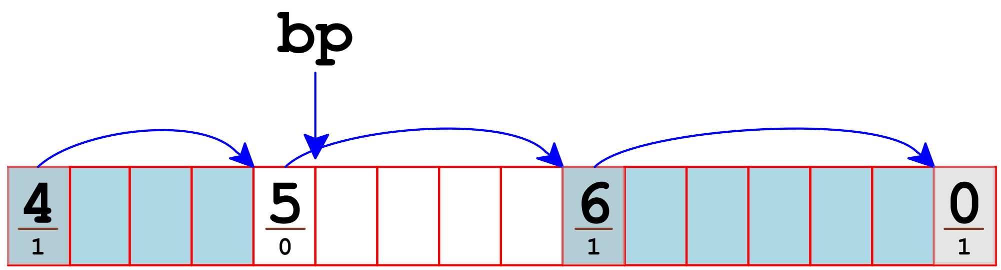
Marking a Block as Allocated




```
void set_allocated(void *bp, size_t size) {
  size_t extra_size = GET_SIZE(HDRP(bp)) - size;

  if (extra_size > ALIGN(1 + OVERHEAD)) {
    GET_SIZE(HDRP(bp)) = size;
    GET_SIZE(HDRP(NEXT_BLKP(bp))) = extra_size;
    GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 0;
  }

  GET_ALLOC(HDRP(bp)) = 1;
}
```

Copy

Marking a Block as Allocated

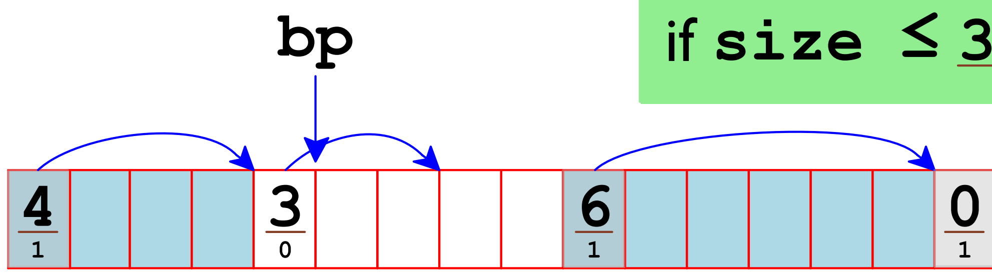



```
void set_allocated(void *bp, size_t size) {
  size_t extra_size = GET_SIZE(HDRP(bp)) - size;

  if (extra_size > ALIGN(1 + OVERHEAD)) {
    GET_SIZE(HDRP(bp)) = size;
    GET_SIZE(HDRP(NEXT_BLKP(bp))) = extra_size;
    GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 0;
  }

  GET_ALLOC(HDRP(bp)) = 1;
}
```

Copy

Marking a Block as Allocated

if `size` ≤ 3: ***split***



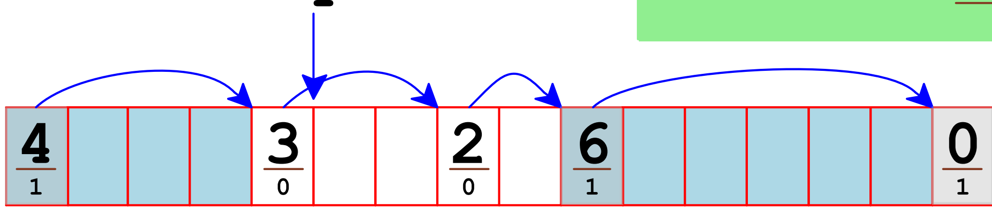

```
void set_allocated(void *bp, size_t size) {
  size_t extra_size = GET_SIZE(HDRP(bp)) - size;

  if (extra_size > ALIGN(1 + OVERHEAD)) {
    GET_SIZE(HDRP(bp)) = size;
    GET_SIZE(HDRP(NEXT_BLKP(bp))) = extra_size;
    GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 0;
  }

  GET_ALLOC(HDRP(bp)) = 1;
}
```

Copy

Marking a Block as Allocated

bp

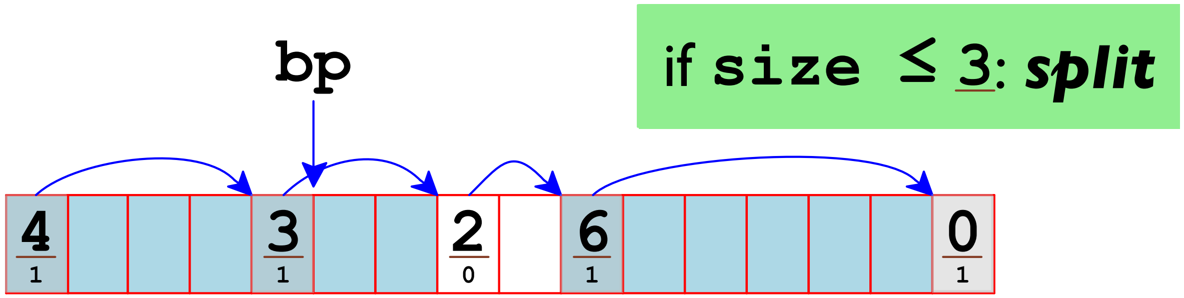
if `size` ≤ 3: ***split***




```
void set_allocated(void *bp, size_t size) {
  size_t extra_size = GET_SIZE(HDRP(bp)) - size;

  if (extra_size > ALIGN(1 + OVERHEAD)) {
    GET_SIZE(HDRP(bp)) = size;
    GET_SIZE(HDRP(NEXT_BLKP(bp))) = extra_size;
    GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 0;
  }

  GET_ALLOC(HDRP(bp)) = 1;
}
```

Copy

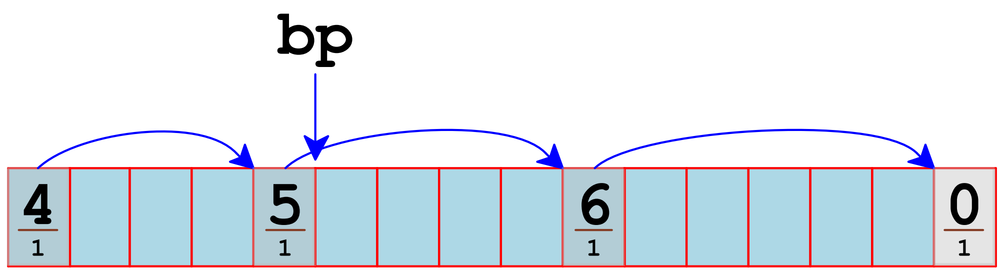
Marking a Block as Allocated




```
void set_allocated(void *bp, size_t size) {
  size_t extra_size = GET_SIZE(HDRP(bp)) - size;

  if (extra_size > ALIGN(1 + OVERHEAD)) {
    GET_SIZE(HDRP(bp)) = size;
    GET_SIZE(HDRP(NEXT_BLKP(bp))) = extra_size;
    GET_ALLOC(HDRP(NEXT_BLKP(bp))) = 0;
  }

  GET_ALLOC(HDRP(bp)) = 1;
}
```

Copy

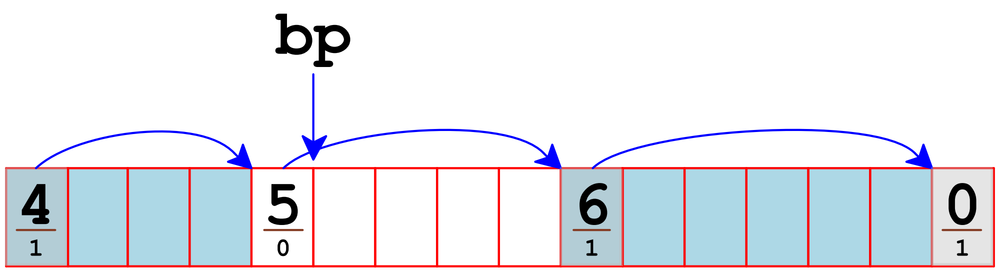
Freeing a Block




```
void mm_free(void *bp) {
  GET_ALLOC(HDRP(bp)) = 0;
}
```

Copy

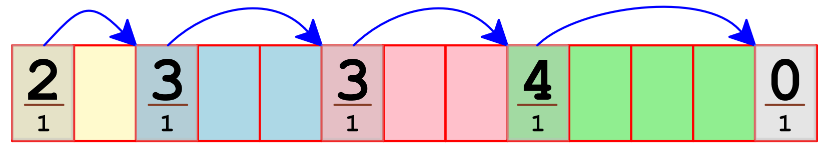
Freeing a Block



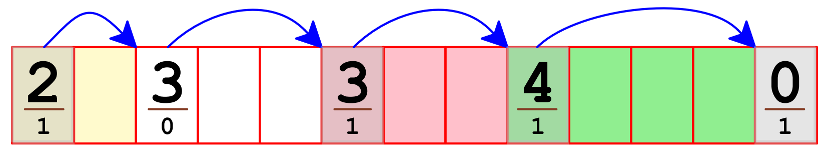

```
void mm_free(void *bp) {
  GET_ALLOC(HDRP(bp)) = 0;
}
```

Copy

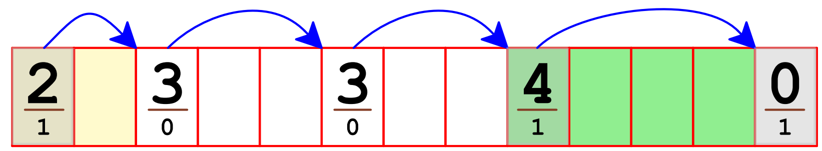
Freeing Multiple Blocks



`free(p2)`

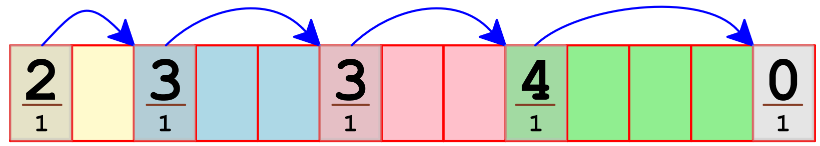


`free(p3)`

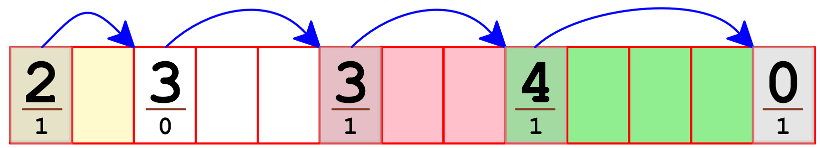


`malloc(5)`

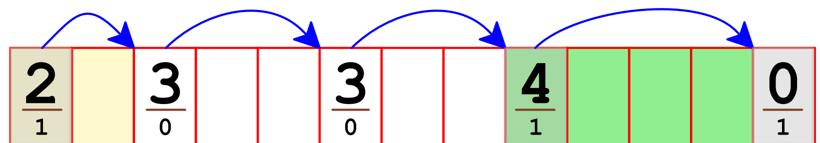
Freeing Multiple Blocks



`free(p2)`



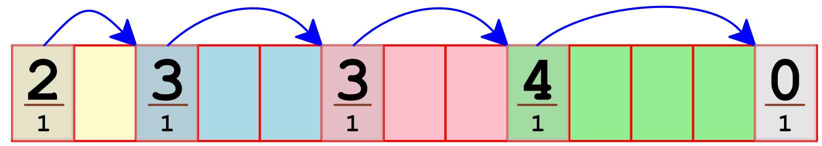
`free(p3)`



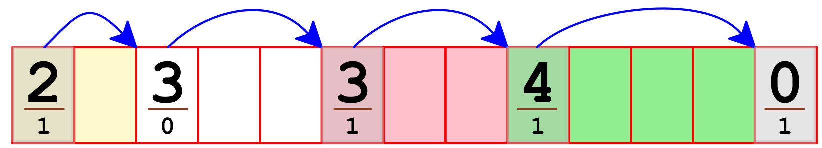
`malloc(5)`

there’s room here, but no unallocated block is big enough ⇒ extra fragmentation

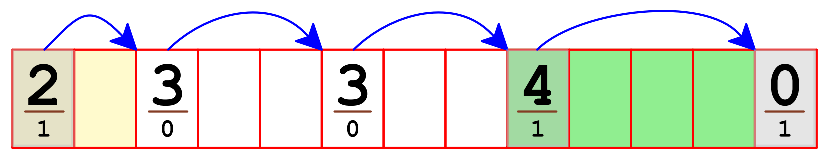
Freeing Multiple Blocks



`free(p2)`



`free(p3)`



`malloc(5)`

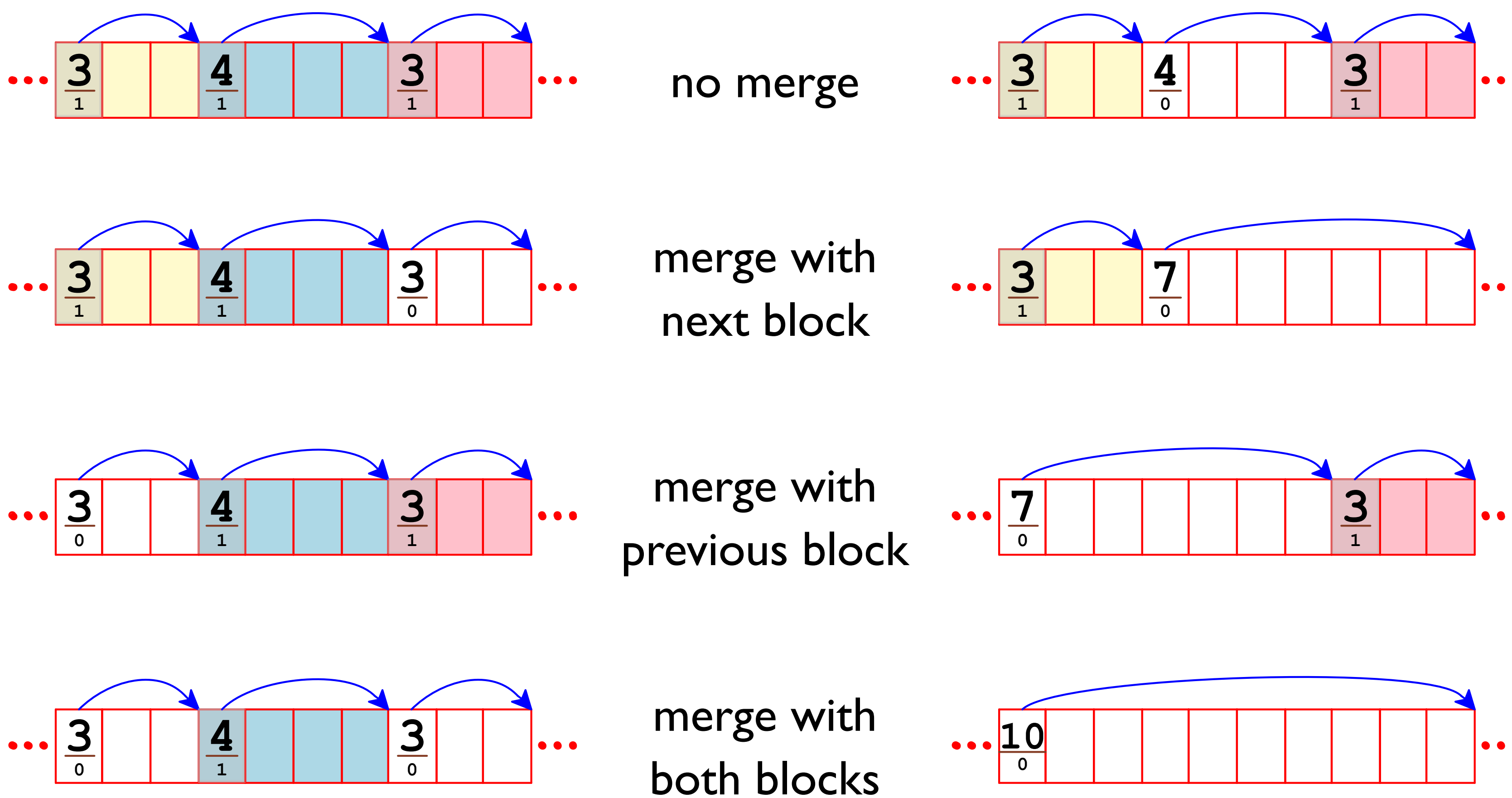
`free` should ***coalesce*** adjacent unallocated blocks

Coalescing Unallocated Blocks

Needed invariant: no two unallocated blocks are adjacent

can maintain at each **`free`** call

For **`free(p2)`**:

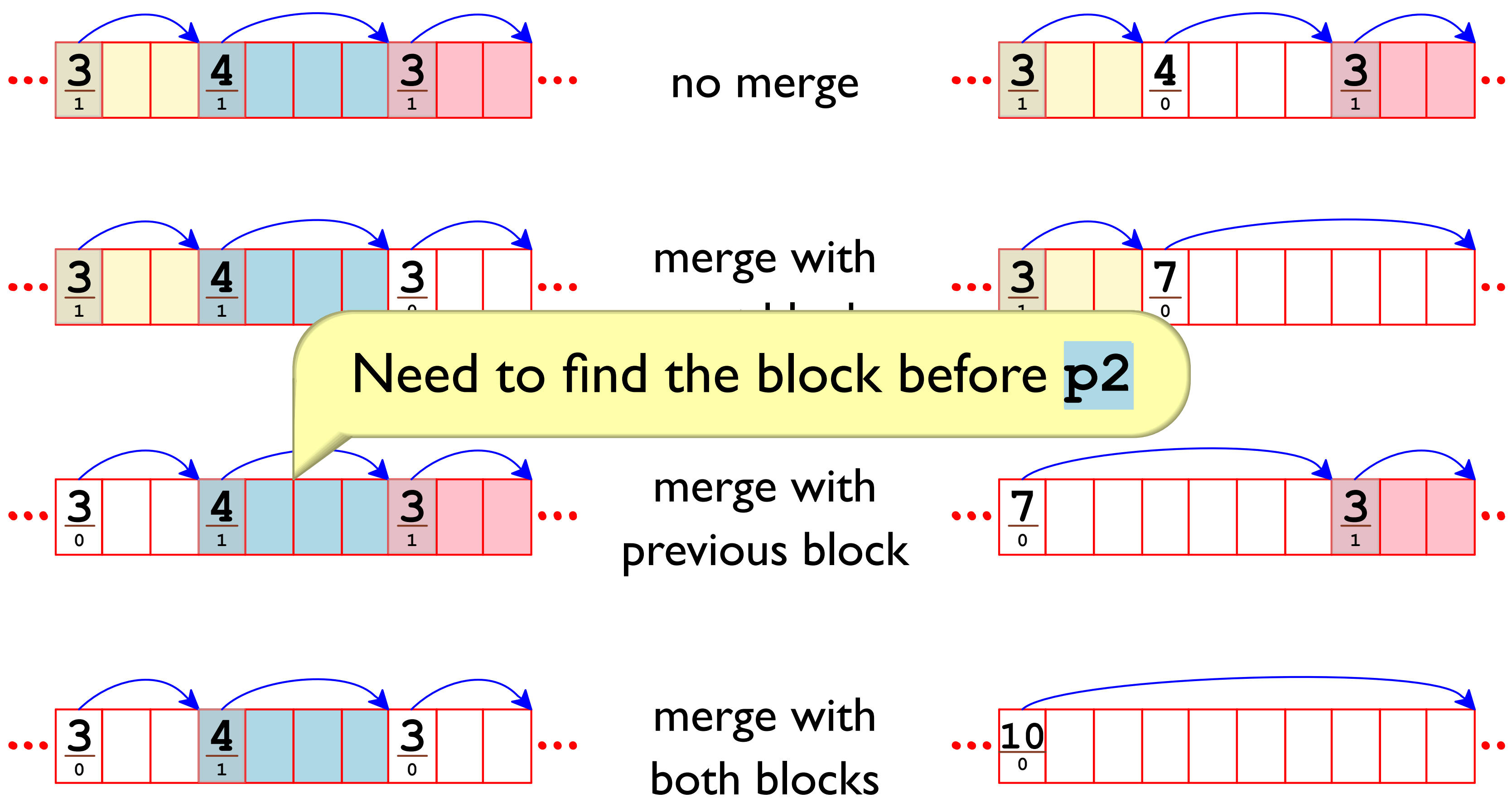


Coalescing Unallocated Blocks

Needed invariant: no two unallocated blocks are adjacent

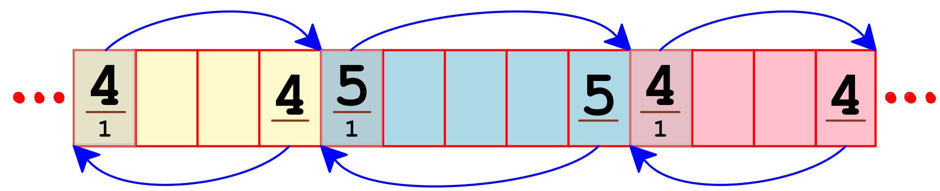
can maintain at each **`free`** call

For **`free(p2)`**:

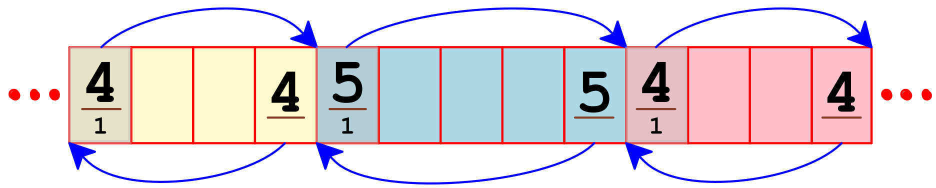


Need to find the block before `p2`

Blocks with Headers and Footers



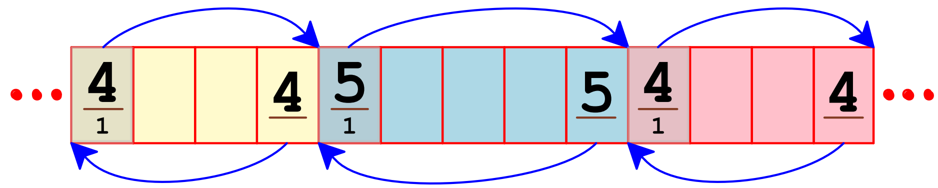
Blocks with Headers and Footers




```
typedef struct {
  size_t size;
  int filler;
} block_footer;
```

Copy

Blocks with Headers and Footers

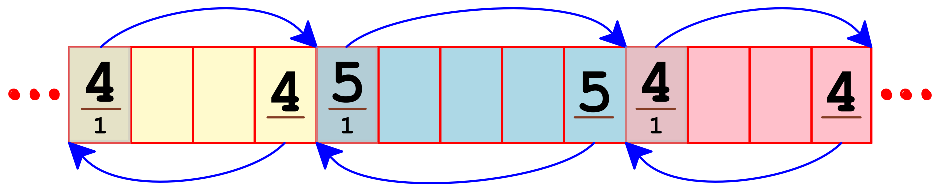



```
typedef struct {
  size_t size;
  int filler;
} block_footer;
```

Same place as in **`block_header`**

Copy

Blocks with Headers and Footers



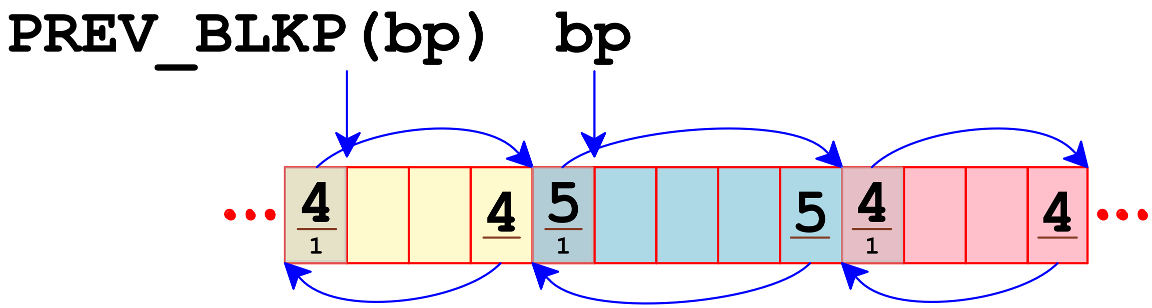

```
typedef struct {
  size_t size;
  int filler;
} block_footer;
```

Copy

```
#define OVERHEAD (sizeof(block_header)+sizeof(block_footer))
```

Copy

Blocks with Headers and Footers



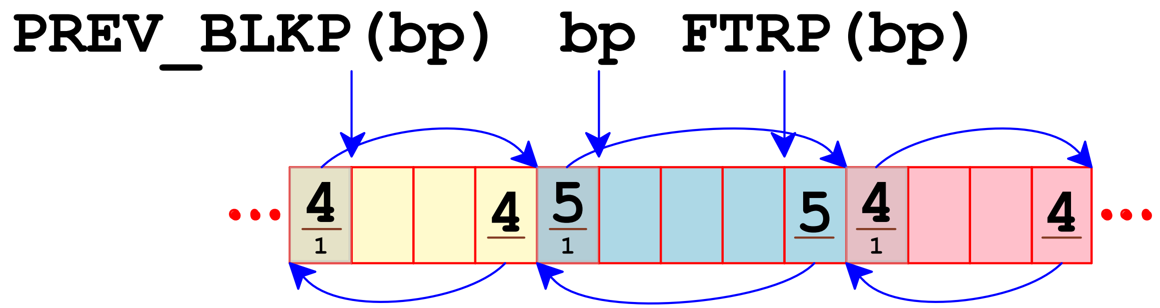

```
typedef struct {
  size_t size;
  int filler;
} block_footer;
```

Copy

```
#define PREV_BLKP(bp) ((char *)(bp)-GET_SIZE((char *)(bp)-OVERHEAD))
```

Copy

Blocks with Headers and Footers




```
typedef struct {
  size_t size;
  int filler;
} block_footer;
```

Copy

```
#define FTRP(bp) ((char *)(bp)+GET_SIZE(HDRP(bp))-OVERHEAD)
```

Copy

Setting Block Sizes in Footers

```
void extend(size_t new_size) {
  ....
  GET_SIZE(HDRP(bp)) = chunk_size;
  GET_SIZE(FTRP(bp)) = chunk_size;
  ....
}

void set_allocated(void *bp, size_t size) {
  ....
    GET_SIZE(HDRP(bp)) = size;
    GET_SIZE(FTRP(bp)) = size;
    GET_SIZE(HDRP(NEXT_BLKP(bp))) = extra_size;
    GET_SIZE(FTRP(NEXT_BLKP(bp))) = extra_size;
  ....
}
```

Copy

Coalescing after Free

```
void mm_free(void *bp) {
  GET_ALLOC(HDRP(bp)) = 0;
  coalesce(bp);
}
```

Copy

Coalescing Free Blocks

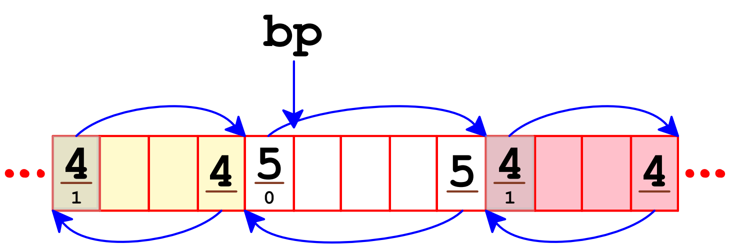
```
void *coalesce(void *bp) {
  size_t prev_alloc = GET_ALLOC(HDRP(PREV_BLKP(bp)));
  size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
  size_t size = GET_SIZE(HDRP(bp));
  ....

  return bp;
}
```

Copy

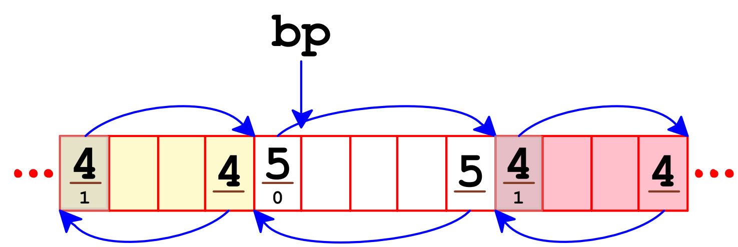
Coalescing Free Blocks

```
void *coalesce(void *bp) {
  size_t prev_alloc = GET_ALLOC(HDRP(PREV_BLKP(bp)));
  size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
  size_t size = GET_SIZE(HDRP(bp));

  if (prev_alloc && next_alloc) {              /* Case 1 */
    /* nothing to do */
  }
  ....
}
```

Copy



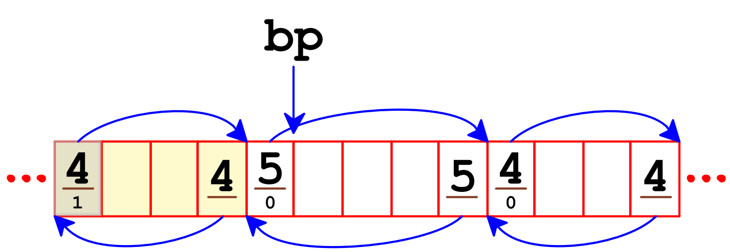
no merge



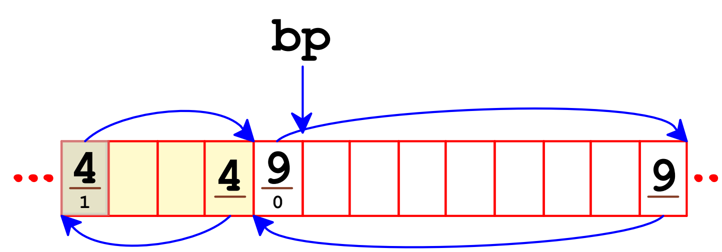
Coalescing Free Blocks

```
void *coalesce(void *bp) {
  size_t prev_alloc = GET_ALLOC(HDRP(PREV_BLKP(bp)));
  size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
  size_t size = GET_SIZE(HDRP(bp));
  ....

  else if (prev_alloc && !next_alloc) {      /* Case 2 */
     size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
     GET_SIZE(HDRP(bp)) = size;
     GET_SIZE(FTRP(bp)) = size;
  }
  ....
}
```

Copy



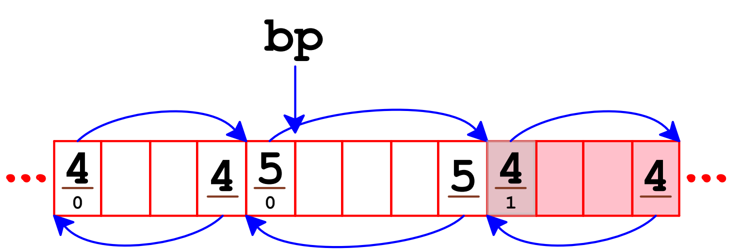
merge with next block



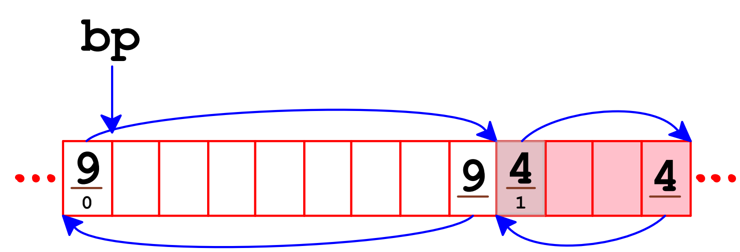
Coalescing Free Blocks

```
void *coalesce(void *bp) {
  size_t prev_alloc = GET_ALLOC(HDRP(PREV_BLKP(bp)));
  size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
  size_t size = GET_SIZE(HDRP(bp));
  ....

  else if (!prev_alloc && next_alloc) {      /* Case 3 */
    size += GET_SIZE(HDRP(PREV_BLKP(bp)));
    GET_SIZE(FTRP(bp)) = size;
    GET_SIZE(HDRP(PREV_BLKP(bp))) = size;
    bp = PREV_BLKP(bp);
  }
   ....
}
```

Copy



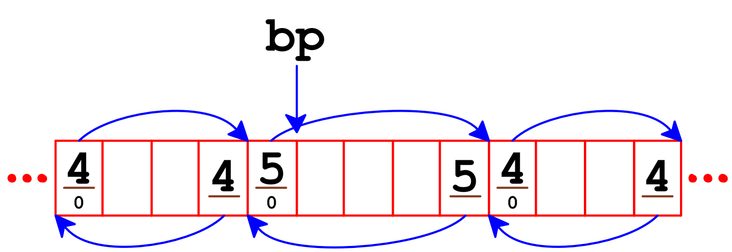
merge with previous block



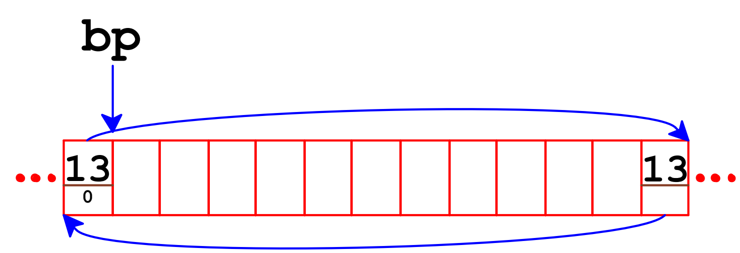
Coalescing Free Blocks

```
void *coalesce(void *bp) {
  size_t prev_alloc = GET_ALLOC(HDRP(PREV_BLKP(bp)));
  size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
  size_t size = GET_SIZE(HDRP(bp));
  ....

  else {                                          /* Case 4 */
    size += (GET_SIZE(HDRP(PREV_BLKP(bp)))
             + GET_SIZE(HDRP(NEXT_BLKP(bp))));
    GET_SIZE(HDRP(PREV_BLKP(bp))) = size;
    GET_SIZE(FTRP(NEXT_BLKP(bp))) = size;
    bp = PREV_BLKP(bp);
  }
  ....
}
```

Copy

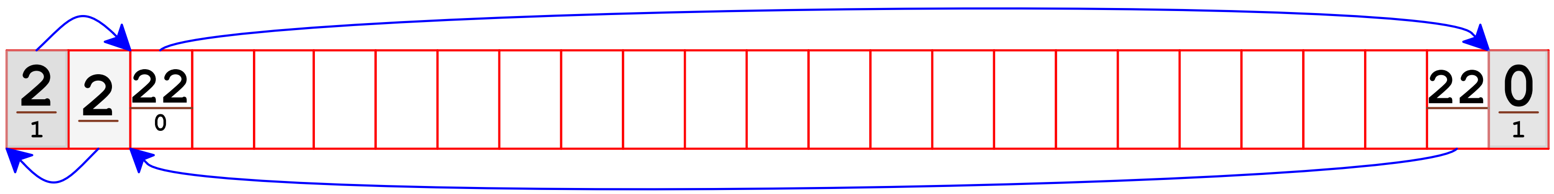


merge with
both blocks



Prolog Block

Create a prolog block so **`coalesce`** can always look backwards




```
int mm_init() {
  ....
 mm_malloc(0); /* never freed */
  ....
}
```

Copy